



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars

VOM FACHBEREICH 18
ELEKTRO- UND INFORMATIONSTECHNIK
ZUR ERLANGUNG DER WÜRDE
EINES DOKTOR-INGENIEURS (DR.-ING.)
GENEHMIGTE DISSERTATION

VON

MSC. ANTHONY ANJORIN

GEBOREN AM

01. NOV 1984 IN ZARIA, NIGERIA

REFERENT: PROF. DR. ANDY SCHÜRR
KORREFERENTIN: PROF. DR. GABRIELE TAENTZER
TAG DER EINREICHUNG: 24.10.2014
TAG DER MÜNDLICHEN PRÜFUNG: 19.12.2014

D17
DARMSTÄDTER DISSERTATION
2015

The work of Anthony Anjorin was supported by the *Excellence Initiative* of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt.

The CME project used as a running example throughout this thesis was funded by the German Federal Ministry of Education and Research, funding code 01|S12054, in the context of the *Software Campus* (www.softwarecampus.de).

The author is responsible for all contents.

Please cite this document as:

URN: `urn:nbn:de:tuda-tuprints-43994`

URI: `http://tuprints.ulb.tu-darmstadt.de/id/eprint/4399`

This document was provided by tuprints, TU Darmstadt E-Publishing-Service:
`http://tuprints.ulb.tu-darmstadt.de`
`tuprints@ulb.tu-darmstadt.de`



This work is licensed under the Creative Commons License:
Attribution-NonCommercial-NoDerivs 3.0 Unported.
<http://creativecommons.org/licenses/by-nc-nd/3.0/>

Anthony Anjorin: *Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars*, © 24.10.2014

ACKNOWLEDGEMENTS

This thesis would not have been possible without the support of numerous people, above all my supervisor *Andy Schürr*, my colleagues *Gergely Varró*, *Marius Lauder*, and *Erhan Leblebici*, and our favourite co-author *Gabriele Taentzer*, who kindly agreed to review this thesis.

Thank you all for the hours and hours of stimulating discussions, Skype sessions, pair-programming sprees, and diagram *kung fu* on the huge white boards in my office!

DECLARATION OF AUTHORSHIP

I warrant that this thesis is my original work. All references and sources that I used have been appropriately acknowledged. I further declare that this work has not been submitted anywhere else for the purpose of academic examination, either in its original, or similar form.

I hereby grant the Real-Time Systems Lab the right to publish, reproduce and distribute my work.

Darmstadt, 24.10.2014

Anthony Anjorin

ABSTRACT

In modern, computer-aided engineering processes, restoring and maintaining the consistency of multiple, related artefacts is an important challenge. This is especially the case in multi-disciplinary domains such as manufacturing engineering, where complex systems are described using multiple artefacts that are concurrently manipulated by domain experts, each with their own established tools.

Bidirectional languages address this challenge of consistency maintenance by supporting incremental change propagation with a clear and precise semantics. Triple Graph Grammars (TGGs) are a prominent rule-based and declarative bidirectional model transformation language with various implementations, and a solid formal foundation based on algebraic graph transformations.

Although TGGs are well suited for synchronizing models that are already on an appropriate, high-level of abstraction, practical model synchronization chains typically require handling models on *different* levels of abstraction. This poses additional challenges including: (i) handling massive information loss typically incurred when abstracting from a low-level model to a high-level model, (ii) supporting complex attribute manipulation as low-level models are often simple trees extracted from textual or XML-files, with relevant information encoded in attribute values rather than structural relations, and (iii) enabling arbitrary structural constraints to cope with complex, often recursive structural context relations, which are usually not present as explicit links in low-level models.

This thesis addresses these challenges by:

- Establishing a general *framework* for organizing and structuring model synchronization chains. This framework is applied to an industrial case study in the domain of manufacturing engineering, which is used consequently throughout this thesis to identify requirements, formulate corresponding challenges, and evaluate the contributions of this thesis.
- Identifying and formalizing new language extensions for TGGs: *attribute conditions* for complex attribute manipulation in TGG rules, and *dynamic conditions* for integrating arbitrary structural constraints. To guarantee the maintainability of large TGG specifications, a new modularity concept for TGGs, *rule refinement* is also introduced. Similar to inheritance and composition for programming languages, rule refinement enables the reuse and flexible combination of TGG rule fragments to form similar TGG rules without introducing redundancy in specifications.
- Extending an existing TGG-based *synchronization algorithm* to cover these new features with formal proofs of correctness and completeness (well-behavedness) of derived TGG-based synchronizers.
- Providing *formal construction techniques and static analyses* for all properties and restrictions required to guarantee the well-behavedness of derived synchronizers.

ZUSAMMENFASSUNG

In modernen, rechnergestützten Prozessen ist es wichtig, die Konsistenz relevanter Artefakte sicherzustellen. Dies ist vor allem in multi-disziplinären Domänen wie der Fertigungstechnik wichtig, in denen mehrere Ingenieure typischerweise gleichzeitig mit verschiedenen Werkzeugen auf diese Artefakte zugreifen.

Bidirektionale Sprachen können durch eine gezielte Propagation inkrementeller Änderungen, die Konsistenz zwischen Artefakten sicherstellen. Tripelgraphgrammatiken (TGGen) sind eine bekannte, regelbasierte und deklarative bidirektionale Modelltransformationssprache mit mehreren Implementierungen und einem soliden, formalen Fundament auf Basis algebraischer Graphtransformationen.

Obwohl TGGen gut geeignet sind, um Modelle auf gleicher, hoher Abstraktionsebene zu synchronisieren, ist es in praktischen Modellsynchronisierungsszenarien oft nötig, auch Modelle auf *unterschiedlichen* Abstraktionsebenen zu synchronisieren. Dies stellt jedoch neue, zusätzliche Anforderungen an TGGen: (i) dem Informationsverlust, der durch den Abstraktionsprozess verursacht wird, muss sinnvoll begegnet werden, (ii) komplexe Attributmanipulation muss gut unterstützt sein, da aus textuellen oder XML-Dateien abgeleitete Modelle häufig Bäume sind, die relevante Informationen vorwiegend in Attributwerten statt in strukturellen Beziehungen kodieren und (iii) es muss möglich sein, flexible strukturelle Bedingungen zu formulieren, da komplexe, oft rekursive, strukturelle Kontextbeziehungen nicht explizit als direkte Verbindungen vorhanden sind.

Diese Arbeit widmet sich diesen Herausforderungen:

- Ein *Framework* für die Organisation und Strukturierung von Modellsynchronisierungsketten wird präsentiert und auf eine industrielle Fallstudie aus der Fertigungstechnik angewendet. Die gleiche Fallstudie wird durchgehend als Beispiel verwendet, um Anforderungen und Herausforderungen abzuleiten sowie die Beiträge dieser Arbeit zu evaluieren.
- Folgende Spracherweiterungen für TGGen werden vorgeschlagen und formalisiert: *Attributbedingungen* für komplexe Attributmanipulation, sowie *dynamische Bedingungen* für die flexible Integration struktureller Einschränkungen. Um die Wartbarkeit großer TGG-Spezifikationen zu gewährleisten, wird *Regelverfeinerung* als neues Modularitätskonzept für TGGen eingeführt. Analog zu Vererbung und Komposition für Programmiersprachen, ermöglicht Regelverfeinerung die Wiederverwendbarkeit und flexible Kombination von TGG-Regelfragmenten.
- Ein existierender *Synchronisierungsalgorithmus* wird um diese neuen Sprach-elemente erweitert, mit formalen Beweisen der Korrektheit und Vollständigkeit abgeleiteter Synchronisierer.
- Es werden *formale Konstruktionsverfahren* sowie *statische Analysen* entwickelt, mit denen alle geforderten Eigenschaften und Einschränkungen geprüft werden können.

CONTENTS

1	CONCURRENT MANUFACTURING ENGINEERING	1
1.1	Overview and Motivation	2
1.2	Stakeholders and Requirements	6
1.3	From Requirements to Challenges and Contributions	10
1.4	Outline and Notational Guide	14
2	FUNDAMENTALS	17
2.1	Models, Metamodels, and Model Transformations	17
2.2	Model Synchronization with Triple Graph Grammars	38
3	A FRAMEWORK FOR MODEL SYNCHRONIZATION	55
3.1	The eMoflon Code Adapter (MOCA) Framework	55
3.2	A TGG-Based Realization of MOCA	58
3.3	Applying MOCA to CME	63
4	SPECIFICATION OF TGG-BASED SYNCHRONIZERS	69
4.1	Flexible Attribute Manipulation in TGG Rules	69
4.2	Additional Structural Constraints with Dynamic Conditions	91
4.3	Modularizing TGGs with Rule Refinement	97
5	DERIVATION OF TGG-BASED SYNCHRONIZERS	121
5.1	Operationalization of TGG-Rules	122
5.2	Precedence-Driven Control Algorithm	137
6	CONSTRUCTION TECHNIQUES AND STATIC ANALYSES	185
6.1	Precedence-Compatibility and Schema-Compliance	185
6.2	Static Analysis of Source Local Completeness	190
6.3	Static Analysis of Forward Local Completeness	198
7	IMPLEMENTATION AND EVALUATION	209
7.1	From Moflon to eMoflon	209
7.2	Core Model Transformations in eMoflon	210
7.3	Scope of Implementation	212
7.4	Evaluation	214
8	RELATED WORK	223
8.1	Frameworks for Model Synchronization	224
8.2	Bidirectional Transformation Languages	229
8.3	Language Extensions for TGGs	234
8.4	TGG-Based Synchronization Algorithms	237
8.5	Restrictions and Static Analyses for TGGs	239
9	CONCLUSION AND FUTURE WORK	243
9.1	A Framework for Model Synchronization	243
9.2	Language Extensions For TGGs	244
9.3	A TGG-Based Synchronization Algorithm and Static Analyses	246
	BIBLIOGRAPHY	249

LIST OF FIGURES

Figure 1.1	Overview of CAD-CAM-CNC process chain	3
Figure 1.2	Example of an incremental update required to support Step 12 in Fig. 1.1	5
Figure 1.3	Stakeholders, subsystems, and main use-cases	6
Figure 1.4	Connecting requirements to challenges and contributions	11
Figure 2.1	Commutative diagrams for associativity and identity con- ditions	18
Figure 2.2	A diagram with objects and arrows in Sets	19
Figure 2.3	Graph morphisms are arrows that preserve the structure in graphs	20
Figure 2.4	Composition of graph morphisms is sound	21
Figure 2.5	Matching a certain GOTO in a chain of GOTOs	22
Figure 2.6	Typed graph morphisms are type preserving graph mor- phisms	23
Figure 2.7	Matching a certain GOTO in a chain of GOTOs in a type pre- serving manner	24
Figure 2.8	Pushout as a generalized union or <i>gluing</i> of objects over a common structure	25
Figure 2.9	Constructing a pushout in Sets	26
Figure 2.10	Diagram used to characterize monomorphisms	27
Figure 2.11	A set of rules for constructing a chain of GOTO commands	28
Figure 2.12	Applying a rule to extend a chain of GOTOs	29
Figure 2.13	Satisfaction of conditions	30
Figure 2.14	Conditions for GOTO graphs and rules	32
Figure 2.15	Meta-levels vs. abstraction levels	34
Figure 2.16	Excerpt of the model space for the GOTO language	36
Figure 2.17	Triple morphisms preserve the triple structure in triple graphs	38
Figure 2.18	A type triple graph and typed triple graph	40
Figure 2.19	Triple rules for consistent GOTO and simulator models	41
Figure 2.20	Deltas for GOTO and simulator models	44
Figure 2.21	Resulting structure produced by TGG synchronizers	45
Figure 2.22	Synchronization results of propagating Δ^1	46
Figure 2.23	Synchronization results of propagating Δ^2	47
Figure 2.24	Alternate synchronization results of propagating Δ^2	48
Figure 2.25	Horizontal and vertical model synchronization	53
Figure 3.1	The eMoflon code adapter (MOCA) framework	56
Figure 3.2	A TGG-based realization of MOCA	58
Figure 3.3	Tree metamodel used as an interface to the EMF modelling space	59
Figure 3.4	Text-to-tree roundtrip for the running example	60
Figure 3.5	Lexer and parser string grammars for the running example	61

Figure 3.6	Tree grammar and string template for the running example	61
Figure 3.7	Required tree-to-model transformation for the running example	62
Figure 3.8	Applying MOCA to the CME case study	64
Figure 3.9	Alternative MOCA-based approach to the CME case study .	66
Figure 4.1	Homomorphisms preserve the structure in algebras	72
Figure 4.2	Attributed graph morphisms are structure preserving maps	73
Figure 4.3	Pushout construction along \overline{M} -morphisms in AGraphs . . .	74
Figure 4.4	Formal and compact notation for attributed graphs	75
Figure 4.5	MocaTree as an attributed type graph	76
Figure 4.6	A match in AGraphs	77
Figure 4.7	A match in AGraphs in compact notation	78
Figure 4.8	A match in ATGraphs in compact notation	78
Figure 4.9	A typed attributed graph over a term extension $A(X)$ of an algebra A	80
Figure 4.10	A direct derivation in ATGraphs	82
Figure 4.11	The rule applied in Fig. 4.10 in compact notation	83
Figure 4.12	TGG schema for the running example	85
Figure 4.13	FileToSimulatorRule	86
Figure 4.14	IgnoreFolderRule	87
Figure 4.15	FirstGotoRule	87
Figure 4.16	IgnoreCommentRule	88
Figure 4.17	GotoGotoRule	89
Figure 4.18	GotoPaintRule	89
Figure 4.19	PaintGotoRule	90
Figure 4.20	Satisfaction of dynamic conditions	92
Figure 4.21	GotoModalPaintRule for handling modal PAINT commands .	94
Figure 4.22	Direct derivation with GotoModalPaintRule	96
Figure 4.23	Roadmap for the following discussion on rule refinement. .	97
Figure 4.24	Syntax of refinements	99
Figure 4.25	<i>DeleteEdge</i> source refinements	99
Figure 4.26	<i>DeleteEdge</i> and <i>CreateEdge</i> source refinements	100
Figure 4.27	<i>ReplaceNode</i> source refinements	101
Figure 4.28	<i>CreateNode</i> source refinements	101
Figure 4.29	<i>ReplaceNode</i> and <i>CreateNode</i> source refinements	102
Figure 4.30	<i>DeleteCorr</i> refinements	102
Figure 4.31	<i>DeleteCorr</i> refinement	103
Figure 4.32	Refinement network for running example	104
Figure 4.33	A coupled grammar is used to derive source components of refinement and corresponding set of source refinement primitives simultaneously	105
Figure 4.34	Coupled grammar ΔG_S for creating source refinement primitives	106
Figure 4.35	Decomposition in target refinement primitives	107
Figure 4.36	Construction of correspondence components and refinement primitives	107

Figure 4.37	Coupled grammar ΔG_C for correspondence components and P_C	108
Figure 4.38	Excerpt of refinement network for running example	109
Figure 4.39	Merge Operator	114
Figure 4.40	Merge operator applied to <code>FirstGotoRule</code> and <code>GotoWith-PrevRule</code>	115
Figure 4.41	Excerpt of refinement network with multi-refinement	118
Figure 5.1	Structure of a source NAC	122
Figure 5.2	Example extended with both allowed and forbidden NACs	123
Figure 5.3	Triple and target delta to illustrate expected synchronizer behaviour	124
Figure 5.4	Derivation of source and forward rules from a TGG rule	126
Figure 5.5	Derivation of source and forward rules from <code>GotoModal-PaintRule</code>	127
Figure 5.6	E-dependency relation and E-concurrent rule	128
Figure 5.7	A TGG rule is an E-concurrent rule of its source and forward rules	129
Figure 5.8	Condition for E-related derivations	129
Figure 5.9	Condition for match consistency	130
Figure 5.10	A pair of match consistent derivations are E-related	132
Figure 5.11	Condition for sequential independence	133
Figure 5.12	Sequential independence of $G_{(k-1)(i-1)} \xrightarrow{sr_k@sm_k} G_{k(i-1)} \xrightarrow{fr_i@fm_i} G_{ki}$	134
Figure 5.13	Sequential independence of $G_{k(i-1)} \xrightarrow{fr_i@fm_i} G_{ki} \xrightarrow{sr_{k+1}@sm_{k+1}} G_{(k+1)i}$	134
Figure 5.14	Derivations ①, ② and ③ used in the proof	136
Figure 5.15	Roadmap for the following sections on precedence-driven synchronization	138
Figure 5.16	Precedence-compatibility and schema-compliance	139
Figure 5.17	Axiom and Goto rules	141
Figure 5.18	Paint rules	142
Figure 5.19	Target negative constraints	143
Figure 5.20	Source negative constraint	143
Figure 5.21	Axiom and Goto rules with NACs	144
Figure 5.22	Paint rules with NACs	145
Figure 5.23	Corrected <code>FirstGotoRule</code>	146
Figure 5.24	Source precedence match and induced source derivation	148
Figure 5.25	Additional context due to fulfilled dynamic conditions	148
Figure 5.26	<code>FirstGotoTargetRule</code> with DEC NACs	150
Figure 5.27	An invalid target precedence match filtered out using a DEC NAC	151
Figure 5.28	Target precedence graph PG_T for target graph G_T	153
Figure 5.29	Relations on precedence graphs and precedence matches used in Alg. 6	154
Figure 5.30	A simple update sequence	157
Figure 5.31	An update sequence that invalidates existing derivations	158

Figure 5.32	Invalidation of precedence match v due to added dangling edge $e : y \rightarrow x$	160
Figure 5.33	Problematic paint rules	165
Figure 5.34	Problematic target graph and target precedence graph . . .	166
Figure 5.35	Local extension $e : SR \rightarrow FL$ of $sm'(SR)$ to forward match $fm(FL)$	168
Figure 5.36	Problematic triple graph and target precedence graph . . .	169
Figure 5.37	Corrected paint rules	170
Figure 5.38	Corrected target precedence graph for forward local complete Paint rules	171
Figure 5.39	Intermediate steps and corresponding states during forward synchronization	177
Figure 5.40	Initial triple graph and source delta	178
Figure 5.41	Triple graph after Step I	178
Figure 5.42	Translation protocol for synchronization	179
Figure 5.43	Precedence graphs for synchronization	180
Figure 5.44	Translation protocol and source precedence graph after Step I	181
Figure 5.45	Finalized source precedence graph	181
Figure 5.46	Synchronized triple graph	182
Figure 6.1	Construction of application conditions from constraints . . .	186
Figure 6.2	Constructing NACs for FirstGotoRule and NoMultiplePaint	188
Figure 6.3	Constructing NACs for FirstGotoRule and NoMultiple-FirstGoto	189
Figure 6.4	Translated graph $TL^* = tr_{TR}(TL)$ in detailed and compact notation	192
Figure 6.5	Target marking rule for GotoPaintXYRule in compact notation	192
Figure 6.6	Forbid-produce and produce-forbid conflicts	195
Figure 6.7	Critical pair for GotoPaintXYRule and GotoPaintXYZRule target marking rules	197
Figure 6.8	Forward local completeness constraint for GotoPaintRule::V0	200
Figure 6.9	Forward local completeness constraint for PaintGotoRule::V0	200
Figure 6.10	Situation used in proof of Thm. 9	204
Figure 6.11	Generated non-trivial positive application condition for FirstGotoRule from the backward local completeness constraint of PaintGotoRule::V0	207
Figure 7.1	Overview of model transformations in eMoflon	211
Figure 7.2	Runtime of batch transformations and backward synchronization	216
Figure 7.3	An excerpt of a TGG rule refinement network used in the CME project.	220
Figure 7.4	A sample TGG rule of average size and complexity.	220
Figure 8.1	Schematic overview of string grammar-based approaches .	225
Figure 8.2	Schematic overview of template-based approaches	226

LIST OF TABLES

Table 1.1	Overview of requirements and affected stakeholders	9
Table 2.1	Basic MDE concepts in the framework of algebraic graph transformations	37
Table 2.2	Interpretation of basic BX concepts in a TGG-based synchronization framework	52

LIST OF DEFINITIONS AND THEOREMS

1	Definition (<i>Category</i>)	17
2	Definition (<i>Sets and Total Functions</i>)	19
1	Fact (Sets is a Category)	19
1	Remark (<i>Totality vs. Partiality</i>)	20
3	Definition (<i>Graphs and Graph Morphisms</i>)	20
2	Fact (Graphs is a Category)	21
4	Definition (<i>Typed Graphs and Typed Graph Morphisms</i>)	23
3	Fact (TGraphs is a Category)	24
5	Definition (<i>Gluing Objects via a Pushout</i>)	25
4	Fact (<i>Pushouts exist in Sets, Graphs, and TGraphs</i>)	26
6	Definition (<i>The Class \mathcal{M} of Monomorphisms</i>)	27
7	Definition (<i>Initial Object</i>)	27
5	Fact (<i>Monomorphisms and Initial Object in TGraphs</i>)	27
8	Definition (<i>Rules, Graph Grammars, and Derivations</i>)	27
9	Definition (<i>Conditions</i>)	30
10	Definition (<i>Constraints and Negative Constraints</i>)	30
11	Definition (<i>Application Conditions and Negative Application Conditions</i>)	31
12	Definition (<i>Graph Languages</i>)	31
13	Definition (<i>Triple Graphs and Triple Morphisms</i>)	38
14	Definition (<i>Typed Triple Graphs and Typed Triple Morphisms</i>)	39
6	Fact (Tri is a Category with Pushouts, Initial Object and Set \mathcal{M} of Monomorphisms)	39
15	Definition (<i>Triple Rules, Triple Graph Grammar</i>)	41
16	Definition (<i>Lifting Standard Operators to TGraphs and Tri</i>)	42
17	Definition (<i>Source, Correspondence and Target Deltas</i>)	43
18	Definition (<i>Consistent Source Delta</i>)	43
19	Definition (<i>TGG-Based Forward/Backward Synchronizers</i>)	45
20	Definition (<i>Formal Properties of TGG-Based Synchronizers</i>)	49
21	Definition (<i>Predicate Algebraic Signature Σ</i>)	70
22	Definition (<i>Predicate Σ-Algebra A_Σ</i>)	71
23	Definition (<i>Predicate Σ-Algebra Homomorphism</i>)	71
7	Fact (Alg (Σ) is a category)	72
24	Definition (<i>Attributed Graphs and Attributed Graph Morphisms</i>)	72
25	Definition (<i>Class $\overline{\mathcal{M}}$ in AGraphs</i>)	73
8	Fact (AGraphs is a category with pushouts along $\overline{\mathcal{M}}$ -morphisms)	74
26	Definition (<i>Final Predicate Σ-Algebra</i>)	75
27	Definition (<i>Typed Attributed Graphs and Typed Attributed Graph Morphisms</i>)	76
9	Fact (ATGraphs is a category with pushouts along $\overline{\mathcal{M}}$ -morphisms in AGraphs)	77

28	Definition (<i>Term Extension $A(X)$ of Algebra A with variables X</i>)	79
29	Definition (<i>Attributed Rules, Attributed Graph Grammars, and Derivations</i>)	80
30	Definition (<i>Typed Attributed Triple Graphs, Typed Attributed Triple Morphisms</i>)	84
10	Fact (ATri is a category with pushouts along $\overline{\mathcal{M}}$ -morphisms)	84
31	Definition (<i>Attributed Triple Rules, Attributed Triple Graph Grammar</i>)	84
32	Definition (<i>Attributed Triple Rules with Attribute Conditions</i>)	84
33	Definition (<i>Dynamic Condition</i>)	91
34	Definition (<i>Refinement</i>)	98
35	Definition (<i>DeleteEdge</i>)	99
36	Definition (<i>CreateEdge</i>)	100
37	Definition (<i>ReplaceNode</i>)	101
38	Definition (<i>CreateNode</i>)	101
39	Definition (<i>DeleteCorr</i>)	102
40	Definition (<i>CreateCorr</i>)	103
41	Definition (<i>AddCondition</i>)	103
42	Definition (<i>Refinement Primitive</i>)	103
43	Definition (<i>Refinement Network</i>)	103
1	Theorem (<i>Soundness of Refinement Decomposition</i>)	110
44	Definition (<i>Merge Operator \oplus</i>)	113
2	Theorem (<i>Merge Operator is Sound</i>)	114
3	Theorem (<i>Completeness of Refinement</i>)	116
45	Definition (<i>Source/Target Negative Application Conditions</i>)	122
46	Definition (<i>Operationalizable TGGs</i>)	122
47	Definition (<i>Source and Forward Rules</i>)	125
48	Definition (<i>E-Dependency Relation and E-Concurrent Rule in Tri</i>)	128
1	Lemma (<i>TGG Rules are E-Concurrent Rules</i>)	128
49	Definition (<i>E-Related Derivations</i>)	129
50	Definition (<i>Match Consistency</i>)	130
2	Lemma (<i>Match Consistent Derivations are E-Related for $i = 1$ and $j = 0$</i>)	131
4	Theorem (<i>Concurrency Theorem</i>)	132
51	Definition (<i>Sequential Independence</i>)	133
3	Lemma (<i>Sequential Independence of Match Consistent Derivations</i>)	133
5	Theorem (<i>Local Church-Rosser Theorem</i>)	134
6	Theorem (<i>Composition and Decomposition Theorem</i>)	135
52	Definition (<i>Precedence-Compatibility</i>)	138
53	Definition (<i>Schema-Compliance</i>)	139
54	Definition (<i>Source Precedence Match Morphism and Induced Source Derivation</i>)	147
55	Definition (<i>Shorthand Notation for Readability</i>)	149
56	Definition (<i>Source Precedence Graph</i>)	152
57	Definition (<i>Relations on Precedence Graphs and Precedence Matches</i>)	153
4	Lemma (<i>Algorithm 6 produces a precedence graph from a precedence graph</i>)	159
58	Definition (<i>Translation Protocol</i>)	161
5	Lemma (<i>Algorithm 7 preserves consistency</i>)	162

59	Definition (<i>Precedence-Induced Derivation</i>)	163
60	Definition (<i>Source Local Completeness</i>)	164
61	Definition (<i>Forward Local Completeness</i>)	167
62	Definition (<i>Local Completeness</i>)	167
6	Lemma (<i>Algorithm 8 preserves consistency</i>)	174
7	Theorem (<i>Synchronization with Alg. 9 is Correct</i>)	182
8	Theorem (<i>Synchronization with Alg. 9 is Complete</i>)	182
2	Remark (<i>On the Efficiency of Alg. 9</i>)	183
3	Remark (<i>On the Incrementality of Synchronization with Alg. 9</i>)	184
11	Fact (<i>Construction of Application Conditions from Constraints</i>)	185
1	Corollary (<i>Construction of Schema-Compliant and Precedence-Compatible TGGs</i>)	187
63	Definition (<i>Source Marking Rules</i>)	190
64	Definition (<i>Forward Marking Rules</i>)	193
7	Lemma (<i>Equivalence of Precedence-Induced and Marking Derivations</i>) . .	193
65	Definition (<i>Confluence</i>)	194
66	Definition (<i>Critical Pair</i>)	194
12	Fact (<i>Sufficient Condition for Confluence</i>)	195
2	Corollary (<i>Sufficient Condition for Source Local Completeness</i>)	196
67	Definition (<i>Forward Local Completeness Constraint</i>)	198
68	Definition (<i>Forward Redundant Negative Target Constraints and Target NACs</i>)	201
8	Lemma (<i>Forward Redundancy</i>)	201
69	Definition (<i>Complementary Rules</i>)	202
70	Definition (<i>Forward Redundant Attribute and Dynamic Conditions</i>) . . .	203
9	Theorem (<i>Sufficient Condition for Forward Local Completeness</i>)	203
4	Remark (<i>Towards a Static Analysis of Local Completeness</i>)	205

The art of building high quality software systems is now considered to be an engineering task comparable to constructing a bridge. We expect software engineers to use an established, standardized development process that guarantees a maintainable end-product at an affordable price.

The Object-Oriented (OO) paradigm has been successful in fulfilling these expectations with high-level language constructs, design patterns, frameworks, middleware, and refactorings. According to Bézivin [14], the success of OO languages can be attributed to the usage of *objects*, units that combine state and functionality, as a unifying concept. In the case of the OO paradigm, the principle of *unification* is realized by regarding everything as an object. This allows for powerful abstractions, required to cope with increasing system complexity.

Model-Driven Engineering (MDE) takes unification a step further by introducing the more general concept of a *model*, an abstraction suitable for a particular task. Regarding everything as a model allows the abstraction process to encompass all engineering artefacts including everything from semi-formal requirements and design documents to executable code. Perhaps even more importantly, MDE places a strong focus on standardizing the software development process. With Object Management Group (OMG) standards such as the Unified Modeling Language (UML) and the Meta-Object Facility (MOF), further challenges can be suitably addressed including systematic reuse, automation via custom code generation, platform independence, and a reduction of the gap between problem and solution domains. A central goal is to enable a model-based *validation* by relevant stakeholders to detect errors as early as possible in the development process.

According to the OMG's Model-Driven Architecture (MDA) vision, a software system is to be ideally developed as a sequence of *model transformations*, mapping in each step, a higher-level model to a more detailed lower-level model, until a usually platform-specific, executable "model" is attained. In addition to the actual process of obtaining suitable abstractions, referred to in MDE as *metamodelling*, the MDA vision implies that the involved model transformations play a central role and are thus crucial for any MDE solution.

As engineering processes are often iterative and concurrent, model transformation chains can seldom be one-way streets. Some form of *round-tripping* between different models must be supported to enable a *bidirectional* exchange of information. Indeed, if models are suitable abstractions, then this implies that different stakeholders require different models, which must co-evolve consistently and be synchronized when necessary.

In this thesis, I shall address the challenge of *model synchronization* in an MDE context by taking state-of-the-art support for bidirectional model transformation languages a decisive step forward. The application domain analysed in this thesis is the domain of Concurrent Manufacturing Engineering (CME). As discussed in [33], concurrent engineering in large, multidisciplinary projects involves substantial coordination and exchange of information among different engineering specialists. Tasks are performed in parallel to speed up the overall process; in CME, for instance, manufacturing engineers are expected to work closely together with product designers. CME as an application domain, therefore, involves:

- Well established and normed engineering processes [25].
- Many different engineers each using their own established tools, models and platforms.
- A clear round-tripping characteristic required to enable iterative cycles in the manufacturing process.
- Finally, as is often the case in real-world scenarios, the “models” involved are typically textual files in established textual languages.

This final characteristic requires an appropriate round-trip between textual artefacts and models in the required standard. This can be generalized to requiring a synchronization of *low-level* models extracted directly from folder structures containing eXtensible Markup Language (XML) files, configuration files, property files, or code in a certain programming language, with *high-level* models already in the required modelling standard and containing only information relevant for the subsequent transformation chain.

In the following, the case study used throughout this thesis to demonstrate the proposed model synchronization framework will be presented in detail. A set of core requirements is derived by analysing the involved stakeholders and the investigated process chain. The primary challenges involved in addressing these requirements are identified, and are mapped to the main results presented in this thesis. This finally yields the problem statement and contribution of this thesis.

1.1 OVERVIEW AND MOTIVATION

A schematic overview of a central process chain in CME is given in Fig. 1.1. The process chain is divided into three clear sub-domains indicated with arrows along the vertical left border of Fig. 1.1. The first domain is Computer Aided Design (CAD), which involves the geometric modelling of a product and focuses on aesthetic and functional properties. The second domain is Computer Aided Manufacturing (CAM), encompassing the actual sequence of steps required to manufacture the product from given raw material. The final domain is Computerized Numerical Control (CNC), handling the generation of executable code that drives a specific family of manufacturing machines.

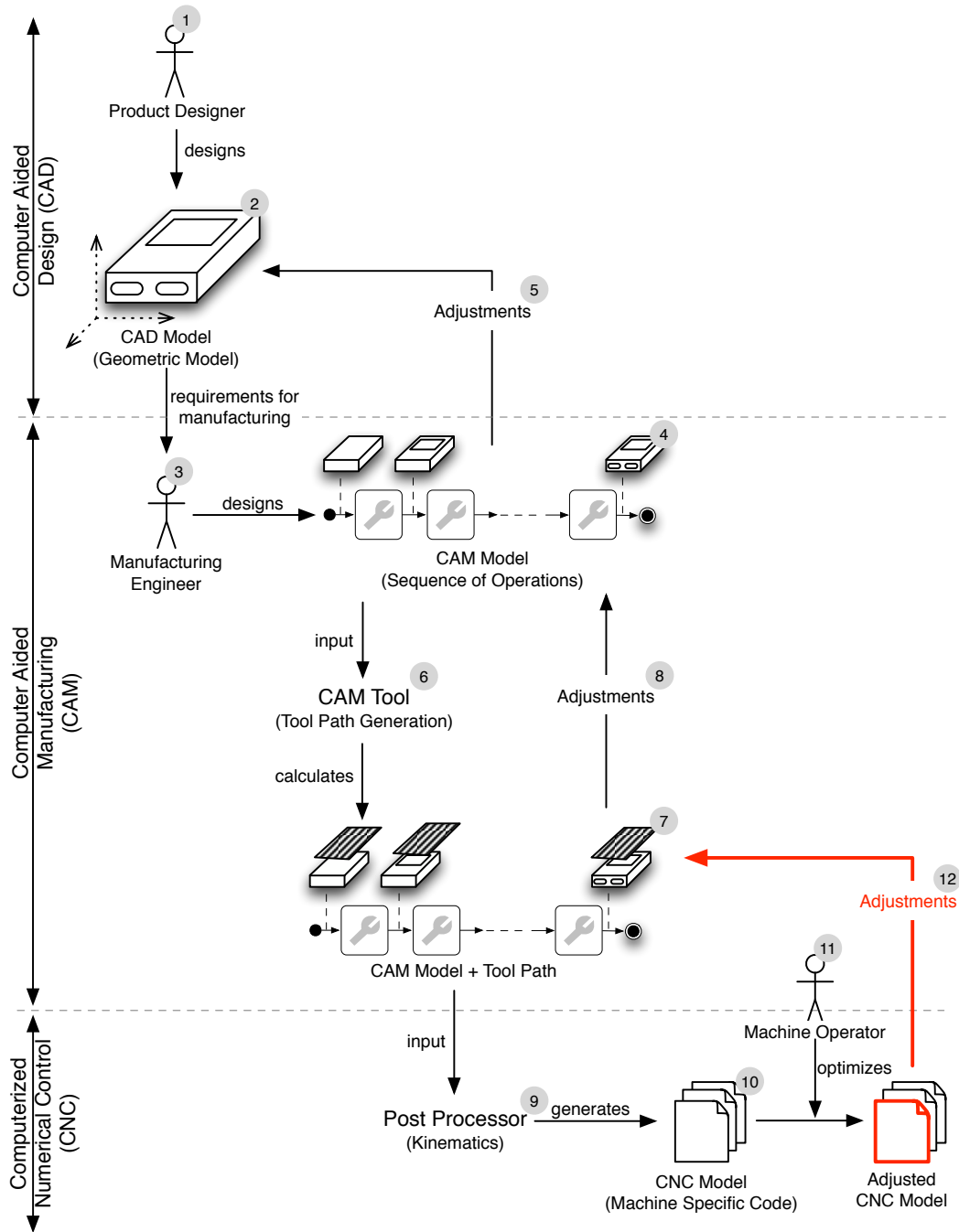


Figure 1.1: Overview of CAD-CAM-CNC process chain

As a first step in the CAD-CAM-CNC process chain, a *product designer* ① specifies a CAD model ②, i.e., a geometric model of a product. This initial CAD model serves as the requirements for a *manufacturing engineer* ③, who specifies a CAM model ④ consisting of a sequence of operations required to manufacture the end-product from the given raw material as specified by the CAD model.

As the CAD model might not be optimal for the manufacturing process, the manufacturing engineer and product designer must work closely together, exchanging information and discussing possible CAM friendly adjustments ⑤ to the CAD model and their consequences. The CAD and CAM models already require a round-trip and synchronization that is, however, currently adequately supported by modern CAM systems, which integrate both CAD and CAM modelling in the same environment.

The specified sequence of manufacturing operations in the CAM model serves as input to a CAM tool ⑥ that calculates the corresponding *tool path* ⑦, the exact movement of the tool tip of the manufacturing machine. Using the generated tool path, the manufacturing engineer can now execute the CAM model in a CAM simulator to further investigate and analyse the manufacturing process. Insights from the simulation are used in this step to further adjust and optimize ⑧ the CAM model. As in the first round-trip between the product designer and the manufacturing engineer, this adjustment cycle based on simulation results is fully integrated in the CAM tool and is thus adequately supported.

Finally, the CAM model with generated tool path is now passed on to a *post processor* ⑨, which generates a CNC model ⑩, i.e., code for a specific family of CNC machines. The generated code is indeed machine family specific as the exact kinematics (geometry of motion) of the machines is exploited to produce highly optimized code.

A *machine operator* ⑪ supervises the manufacturing process and can decide to intervene in the process and further optimize the CNC code. As an example, the machine operator might be more familiar with the exact manufacturing machine at hand and realize that certain operations can be further accelerated. As the CNC development environment is not yet integrated in typical CAM tools, these changes made directly to the generated CNC code must be manually translated into corresponding adjustments ⑫ to the CAM model. This final round-trip between CAM and CNC models is, therefore, highly problematic and is the focus of our case study.

Figure 1.2 depicts the relevant steps in the process chain for a concrete example of an incremental update. For the actual synchronization, a textual export format of the CAM model, a Cutter Location Source (CLS) file, is used. This is depicted as *v0.cls* in the top left corner of Fig. 1.2. The CLS file depicts the CAM model as a series of commands, one on each line. Comments start with \$\$, while all other commands are of the form:

```
command [ "/" argument ("," argument)* ]
```

In *v0.cls* three command types are used: (1) a GOTO command used to position the tip of the machine with x, y, z coordinates as arguments, (2) a PAINT command, and (3) END-OF-PATH used to terminate the program. The PAINT command makes sense as CLS files can be executed by a *simulator* provided by the CAM tool. The simulator not only visualizes the manufacturing process but also displays the tool path specified by the sequence of GOTO commands in the CLS file using the colours set by the PAINT commands.

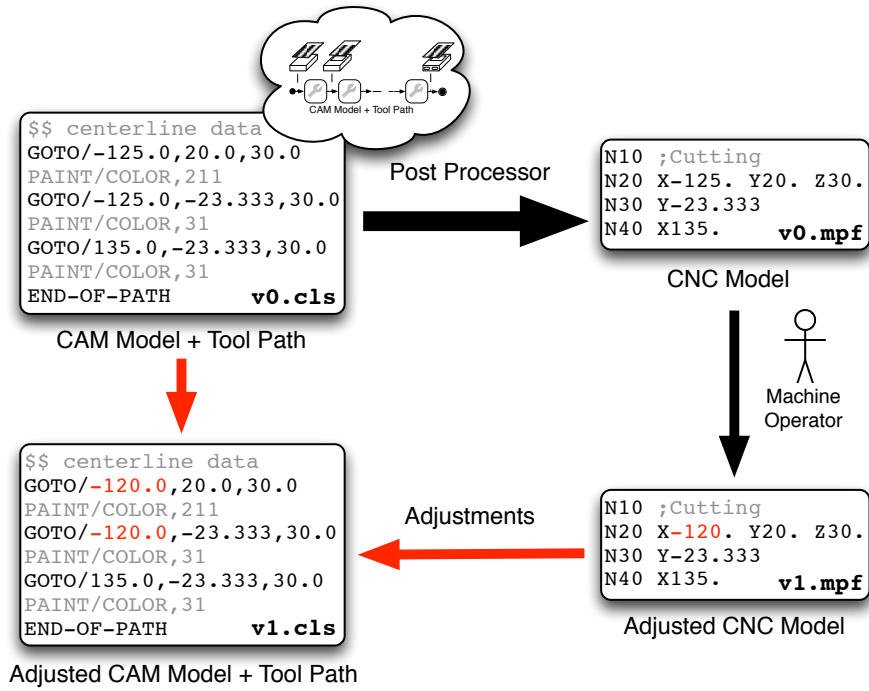


Figure 1.2: Example of an incremental update required to support Step 12 in Fig. 1.1

The CLS file *v0.cls* is transformed by the *post processor* to CNC code in the Main Program File (MPF) *v0.mpf*. CNC code contains comments starting with a semicolon (;), and each line is prefixed with a line number. In *v0.mpf* the three lines below the comment correspond to the three GOTO commands in *v0.cls*. The important points to note are the following:

1. The comments and PAINT commands in the CLS file are irrelevant for the CNC code. This loss of information is present in both directions as, for example, the comments in CNC code are also irrelevant for the CLS file.
2. CNC code is optimized for efficient interpretation by manufacturing machines by only repeating argument values that actually change from command to command. On Lines N30 and N40 of *v0.mpf*, for example, only the y and x coordinates are changed, respectively, and must be specified. All other coordinate values remain the same and do not have to be repeated as in the CLS file. In this context, this is often referred to by domain experts as *modality*, i.e., the commands in CNC code are modal with respect to argument values. An important point here is that the degree of modality in CNC code can be freely determined by a CNC programmer or, as in our example, by the post processor. Although a high degree of modality is preferable for efficiency reasons, certain argument values can be repeated redundantly either to increase readability, or to ease code generation by enabling the reuse of the same set of templates in different contexts. The latter explains why CNC code generated by the post processor is not always as modal as it could be.

In the simple example depicted in Fig. 1.2, the machine operator supervising the manufacturing process decides to optimize the first step in the tool path by changing the x coordinate value from -125 to -120, resulting in the new file v1.mpf. This might be a machine-specific optimization that, for instance, accelerates the process without any negative effects.

The challenge is now to reflect this change in a corresponding CLS file v1.cls, without losing any information from v0.cls. The correct result is depicted in Fig. 1.2 as v1.cls. The following points should be noted:

1. The single change in v1.mpf resulted in *multiple* changes in v1.cls. In this case, this is due to the modality of the commands in v1.mpf, i.e., changing the value of the x coordinate influences all subsequent commands until the value of the x coordinate is changed.
2. The unaffected commands in v0.cls are retained unchanged in v1.cls, e.g., comments, PAINT commands, and unaffected GOTO commands. As it is impossible to conjure up commands, e.g. PAINT, that are irrelevant for CNC code given only v1.mpf, this implies that the update must be realized by somehow using v0.cls as well. A possibility is to *incrementally* update v0.cls to produce v1.cls.

1.2 STAKEHOLDERS AND REQUIREMENTS

After discussing a small concrete example, we are now ready to analyse all stakeholders involved in the application scenario in more detail and induce a set of requirements, which must be considered when making design choices (cf. Fig. 1.3).

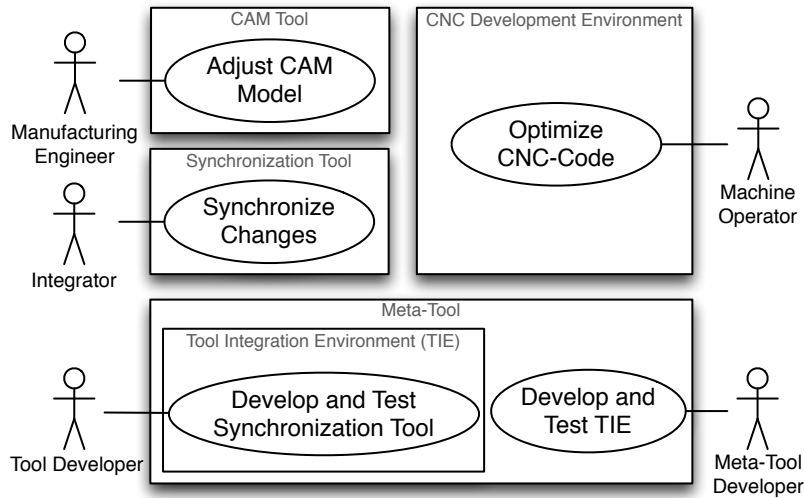


Figure 1.3: Stakeholders, subsystems, and main use-cases

THE MANUFACTURING ENGINEER is in charge of creating and maintaining the CAM model in a CAM tool (Fig. 1.3), and is able to translate changes made to the tool path in a CLS file to the actual sequence of operations in the CAM model. To explain this using our simple example, the resulting `v1.cls` can be loaded into a CAM tool and visualized. This is, however, only an indication of the changed tool path and cannot be used to automatically adjust the sequence of operations and chosen tools in the CAM model. This is considered out-of-scope for the synchronization and must be realized by the manufacturing engineering using the change information.

Apart from requiring that the changes be plausible, the manufacturing engineer expects to be able to load the changed CLS file in the CAM tool to visualize the tool path. This means that textual files conforming to the CLS standard must be generated by the synchronization tool (Fig. 1.3).

↪ In more general terms, *text generation or unparsing (R1)* is required. In contrast, an editor or other form of visualization of CLS files is not required.

THE MACHINE OPERATOR supervises the manufacturing process and optimizes the CNC code in the normal CNC development environment (Fig. 1.3), which already provides adequate editors and visualizations for CNC code.

↪ This means that the changes made are provided as two CNC files, before the change and after (e.g., `v0.mpf` and `v1.mpf`) inducing two requirements: *parsing of CNC code (R2)*, and a *diff for change detection (R3)*.

THE INTEGRATOR, normally the same person as the manufacturing engineer, needs to understand the consequences of changes made by the machine operator to decide if the adjustments are to be accepted or rejected.

↪ To ease the comparison of different CLS files (e.g., `v0.cls` and `v1.cls`) this requires the *preservation of all unaffected commands, comments, and layout (R4)*.

The integrator also needs to plan and explore other possibilities of reconciliation by applying changes this time in CLS files and considering the consequences in CNC code.

↪ This is only possible if the synchronization is *bidirectional (R5)*.

The integrator works with the provided synchronization tool and performs multiple updates in both directions in the course of reconciling CLS files with CNC code.

↪ To be able to work productively with the tool, it must be reasonably *efficient (R6)* with regards to runtime and memory consumption, and *scale (R7)* for model sizes used in practice.

In the process of reconciliation, the integrator needs to explore different solutions and must be able to weigh and compare different design decisions with the synchronization tool, i.e., without changing the underlying transformation.

↪ This means that the bidirectional transformation must capture and preserve all degrees of freedom, i.e., *a means for configuration and user interaction (R8)* is required.

In general, the integrator must reconcile changes resulting from *concurrent* engineering activities, meaning that potentially *conflicting* changes made in *different* domains to *different* models must be handled.

↪ This requires *support for automatic conflict detection and resolution (R9)*.

THE TOOL DEVELOPER works in a Tool Integration Environment (TIE) that provides support for developing synchronization (tool integration) solutions. Such a TIE is provided by a *meta-tool*, a tool for developing tools, as depicted in Fig. 1.3. In practice, the tool developer and integrator can be the same person, especially in the test phase of a developed synchronization tool.

First and foremost, the tool developer must be able to express the transformation with the provided transformation language.

↪ This means that the transformation language must be *expressive (R10)* enough for practical scenarios.

↪ A potentially opposing requirement to expressiveness is *productivity and maintainability (R11)*, justifying the need for a Domain Specific Language (DSL) for developing synchronization solutions.

↪ As the tool developer might be new to the TIE, a thorough *validation and static analysis (R12)* of specifications made in the provided transformation language is a crucial requirement for being able to work in the TIE.

↪ Finally, the tool developer requires a standard recipe or *framework (R13)* for solving the same class of problems addressing typical requirements from the integrator and other involved domain experts.

THE META-TOOL DEVELOPER uses a meta-tool (Fig. 1.3) to develop the TIE for the tool developer. Although the meta-tool and tool developer can be the same person, especially in an academic context, this is hopefully not the norm! As the meta-tool is a *tool* for developing *tools*, a further requirement, which stems from the MDE unification principle (everything is a model), is that the meta-tool should also be a meta-meta-tool. This enables iterative *bootstrapping*, using a tool to build the next version of itself,¹ a well-known technique from compiler construction. It is important to note that this is not an abstract notion or exotic requirement with little practical relevance. Bootstrapping is indeed a core requirement and the potential of a meta-tool

¹ How bootstrapping is applied in this thesis is handled in Chap. 7 on implementation details.

is greatly reduced if “internal” metamodels and data structures are handled differently from “normal” metamodels [14].

- ↪ The meta-tool developer requires a mature and constructive *theoretical foundation* (R14) for implementing the TIE, as well as *an established process, standards and conventions* (R15) based on which appropriate tool support can be provided.

Table 1.1 gives a summary of the identified requirements and affected stakeholders. Note that the discussion of requirements up to this point has been independent of a particular solution domain.

	Manufact. Eng.	Machine Op.	Integrator	Tool Dev.	Meta-Tool Dev.
Text generation, unparsing (R1)	✗				
Text parsing (R2)		✗			
Change detection (Diff) (R3)		✗			
Information preservation (R4)			✗		
Bidirectionality (R5)			✗		
Efficiency (R6)			✗		
Scalability (R7)			✗		
Configuration and User Interaction (R8)			✗		
Conflict Detection and Resolution (R9)			✗		
Expressiveness (R10)				✗	
Productivity and Maintainability (R11)				✗	
Validation and Static Analysis (R12)				✗	
Framework (R13)				✗	
Theoretical Foundation (R14)					✗
Established Standards/Conventions (R15)					✗

Table 1.1: Overview of requirements and affected stakeholders

1.3 FROM REQUIREMENTS TO CHALLENGES AND CONTRIBUTIONS

In the following, requirements R1 – R15 are grouped together to formulate a set of challenges that any solution must address. The resulting overview of the connection between requirements² and challenges is depicted in Fig. 1.4.

CHALLENGE I: STANDARDIZATION

R13 demands a general framework for structuring model synchronization scenarios. R3 and R15 demand a standardized approach, in which existing diff mechanisms can be integrated. R1 and R2 demand handling of textual content in the framework, and can be generalized to being able to handle XML files, as well as structures in a file-system (file and folder hierarchies). All these structures are basically simple trees for which mature “(un)parsers” exist. Such a framework must thus provide a means of integrating existing (un)parsing technology. R4, R5 and R8 demand an incremental, bidirectional, and configurable transformation language, respectively.

- ↪ The challenge here is to combine a Bidirectional Transformation (BX) language with standard parser, unparser, and diff technologies in a systematic, standardized fashion, allowing for a (partial) reuse of generic adapters and existing solutions.

CHALLENGE II: PRODUCTIVITY AND EXPRESSIVENESS

On the one hand R6, R7 and R10 demand expressiveness *and* efficiency/scalability, which are crucial requirements for tackling real-world applications. On the other hand R11 demands productivity and maintainability, which can only be addressed with a suitably high-level BX language.

- ↪ The challenge here is to provide a transformation language that is at the same time high-level enough to support productivity and maintainability, and is also expressive, efficient and scalable enough for real-world applications. These are, however, potentially contradicting challenges that demand a careful compromise.

CHALLENGE III: FORMAL UNDERPINNING

Finally, R9, R12 and R14 demand a constructive theoretical foundation, and an automated means of checking that all conditions and *well-formedness* constraints are satisfied for specifications made in the provided transformation language, if possible at compile time. Examples for well-formedness constraints, which depend on the transformation language and underlying formalism, include:

- Restrictions that the underlying transformation engine requires to guarantee a worst-case runtime efficiency, e.g., polynomial (and not exponential) runtime in the size of models. This could mean that certain language features or combinations thereof are potentially very inefficient, e.g., require some form of backtracking, and are to be best avoided in specifications.

² Requirements that are out-of-scope for this thesis are depicted in red (grey in a monochrome printout) in Fig. 1.4

- Sufficient conditions, typically also restricting the usage of certain language features, that guarantee that the transformation process terminates, does not abort prematurely with an error, and produces “correct” results, as defined for the concrete transformation language and formalism.
 - Laws that a specification must obey to guarantee *compositionality* with other specifications.
- ↪ The challenge here is to provide a constructive, coherent formal underpinning of all concepts and language constructs, which can serve as a guide for a corresponding implementation and static analysis.

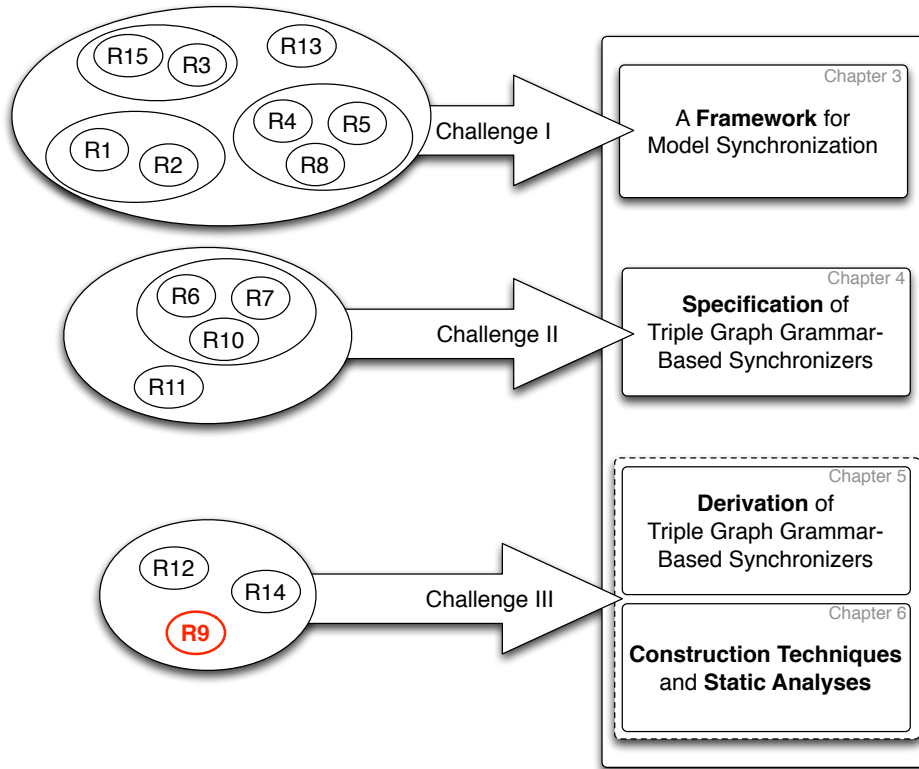


Figure 1.4: Connecting requirements to challenges and contributions

We are now ready to list the contributions of this thesis in connection with the identified challenges, and to justify the chosen solution space. The connection between challenges and contributions is depicted in Fig. 1.4. In the following sections, each contribution is presented by first rephrasing the addressed challenge in the chosen solution space for this thesis (concrete approach, standard, formalism, and technology), before characterizing the contribution correspondingly in the given context.

CONTRIBUTION I: To appropriately address **CHALLENGE I** in this thesis, a thorough MDE approach is taken, using a de-facto standard modelling language *Ecore*, which is isomorphic to a subset of MOF. Furthermore, a strong focus is placed on the involved model transformations, handling the involved artefacts (textual in the case of CME) with standard parser and unparser technology. The OMG provides Query View Transformation (QVT) as a standard for bidirectional model transformation, and I shall use and extend Triple Graph Grammars (TGGs) in this thesis, which can be regarded (to a certain extent) as an implementation and further development of the QVT standard [45]. TGGs, initially introduced by Schürr [88], have been used successfully for various industrial and academic projects [41, 46].

From our experience, e.g., with a recent industrial project [85] in the domain of automation engineering, **CHALLENGE I** is currently addressed individually for each project without applying a systematic approach and possibly re-using generic adapters and solutions. In the worst case, the “tool adapters” to be implemented can actually be more complex than the TGG used to synchronize the extracted models. Shifting most of the complexity to these “auxiliary” transformations implemented by such (tool) adapters is not only unproductive, but also defeats the usage of an incremental model synchronization algorithm, which is blind to these parts.

↪ In this thesis, therefore, a TGG-based framework for designing model synchronization chains in a systematic manner is established and discussed in Chap. 3.

CONTRIBUTION II: TGGs are a rule-based DSL for bidirectional model transformation and are consequently limited in expressiveness. Although this is normal and indeed an inherent characteristic of a useful DSL [37], TGGs do not currently have a well-defined means of falling back to a more expressive host language (possibly a General Purpose Language (GPL) such as Java). This is required for complex and often problem-specific *attribute manipulation*, as well as for expressing complex, possibly recursive *context relationships* in TGG rules.

Currently, the limited expressiveness of TGGs is overcome either via a pre- and post-processing step, or by using an additional language such as the Object Constraint Language (OCL). The former defeats any support for synchronization (which is again blind to these parts), shifts the complexity and focus away from the TGG specification, and is highly unproductive as the necessary model traversal must be specified redundantly for pre-processing, for post-processing again, and of course in the TGG itself. The latter defeats any formal analysis of the specification and adds complexity with an additional, arguably equally or even more expressive language.

↪ In this thesis, therefore, **CHALLENGE II** is addressed by providing a controlled, well-defined, and suitably restricted *integration* of a GPL in TGGs. This means that GPL modules can be integrated into TGG rules either to increase expressiveness, or to optimize parts of the rule

for efficiency. Furthermore, a novel TGG language feature *rule refinement* is introduced with which recurring parts of TGG rules can be reused. This not only boosts productivity as these parts do not need to be specified repeatedly, but also improves maintainability of a large TGG. These new TGG language extensions are presented in Chap. 4.

CONTRIBUTION III: TGGs are a rule-based technique for specifying bidirectional model transformation and are based on the mature formalism of algebraic graph transformations [28]. The solid formal foundation for TGGs is a clear advantage that is crucial for applying static analysis techniques and proving the soundness of required TGG language extensions.

Consequently, an algebraic approach according to [28] is taken, which is based on category theory. The chosen formalism is not only suitable for formalizing MDE concepts that tend to be close to OO concepts, but is also constructive in nature, providing constructive definitions, arguments and proofs that lead almost directly to an implementation [86]. Models are formalized as typed, attributed graphs, while model transformation is formalized via a generalized subtraction and union of graphs.

All TGG tools require a set of additional constraints and conditions to guarantee runtime efficiency (polynomial and not exponential runtime with respect to model size). Most TGG tools do not, however, provide an automated static analysis for actually checking that these conditions are satisfied. In many cases, even the formal basis for implementing such static analyses is still missing.

↪ In this thesis, therefore, an existing TGG-based synchronization algorithm is extended in Chap. 5 to support all new TGG language features together with formal proofs for correctness and completeness under certain conditions. The existing formal foundation for TGGs is consequently extended in Chap. 6 to provide a design-time validation of these required conditions. This contribution enables an *operationalization* of TGGs with all new language features, i.e., an automatic derivation of TGG-based synchronizers that are efficient and well-behaved with respect to the underlying TGG specification.

Conflict detection and resolution (R9) is, however, out-of-scope for this thesis and is a current research focus with both theoretical and practical open research questions. In the CME case study, therefore, only sets of changes in *one domain* can be synchronized in a single step.

Based on Fig. 1.4, the research question to be investigated in this thesis can be summarized as the following hypothesis:

Hypothesis

Can a TGG-based framework for *model synchronization* be established and used effectively for real-world applications, which require support for *both* high-level and low-level models?

I shall show in this thesis that the answer to this question is definitely *yes*.

1.4 OUTLINE AND NOTATIONAL GUIDE

To ensure that every reader has the necessary (formal) background to understand the rest of the thesis, the following Chap. 2 presents a basic formalization of MDE and TGG concepts. Together with the case study representing a synchronization scenario in the domain of CME already presented in Chap. 1, this sets the stage for all other following chapters.

A TGG-based framework for designing model synchronization chains in a systematic manner is established and discussed in Chap. 3. A set of new TGG language features that together achieve a well-defined and careful compromise between formal guarantees (such as completeness) and expressiveness of TGGs, is identified and formalized in Chap. 4. To complete the main contribution of this thesis, Chap. 5 and Chap. 6 provide a formal underpinning for the operationalization of TGG specifications, derivation of TGG-based synchronizers, and static analyses of all restrictions required for a successful derivation and synchronization process.

As a proof-of-concept, all proposed TGG language extensions were implemented in the metamodeling and model transformation tool eMoflon. An overview of this implementation together with technical details and an evaluation of runtime scalability is provided in Chap. 7.

Chapters 8 and 9 conclude this thesis with an overview of, and critical comparison to related approaches, a summary of the main results in this thesis, and an outlook on ideas for further improving support for bidirectionality in general, and TGGs in particular.

Throughout the thesis, the following notational conventions are used:

- The first occurrence of important terms is depicted in *italics*. Italics are, however, also used as a general form of emphasis, comparable to raising one's voice when reading the text aloud.
- When referring to entities in code, a mono-spaced font is used.
- Words are set in "apostrophes" to highlight them and improve readability. This is sometimes necessary when a certain term is used in the current context with a different meaning than is normally the case.
- Every acronym is spelled out *in situ* the first time it is used and is from then on linked to its definition in the complete list of acronyms on Page 261.
- References such as "Fig.", "Sect.", and "Chap." are always abbreviated except when starting a sentence.
- British + ize is used consequently, following the spelling preferred by the (Concise) Oxford English Dictionary (language tag en-GB-oed).
- Although all diagrams are optimized for a coloured print-out or electronic viewing, nothing crucial is lost in a monochrome print-out.

- Two types of diagrams are used in this thesis: (1) informal diagrams such as all figures in this chapter (e.g., Fig. 1.1), and (2) formal diagrams that have a precise categorical semantics (e.g., Fig. 2.1). The latter are always framed and labelled with the category in which they are to be interpreted (from which the objects and arrows are taken).
- First order logic is used in formal definitions and theorems, when a precise logical formulation is required. Note that only commas “,” (meaning *such that* or *the following holds*) are used to link logical expressions:

$$\forall b \in B, \exists a \in A, f(a) = b$$

\Leftrightarrow *for all b in B, there exists an a in A, such that f(a) = b.*

Colons “:” which are sometimes used instead are avoided as this is reserved for denoting arrows such as $r : L \rightarrow R$.

Finally, [] is used to group logical expressions in cases where this improves readability.

FUNDAMENTALS

This chapter is based on the following books [28, 83, 86] and introduces all basic concepts required to understand the rest of this thesis. To improve readability, numerous examples from the CME case study (cf. Chap. 1) are used to impart an intuitive understanding. The only assumption made is that the reader is familiar with basic set theory.

2.1 MODELS, METAMODELS, AND MODEL TRANSFORMATIONS

Our first goal is to formalize the MDE concepts *model*, *metamodel* and *model transformations*. Although this can be done set-theoretically or with a logic-based approach, we shall take an algebraic approach, based on basic category theory.

2.1.1 Sets, Graphs, and Typed Graphs

On the way to defining more complex structures (e.g., for models) we shall start by defining what we want to regard as the simplest possible structure, a *category*.¹

Definition 1 (*Category*).

A category $\mathbf{C} = (\text{Ob}, \text{Arr}, \circ, \text{id})$ consists of:

- a class Ob of *objects*,
- for each pair of objects $A, B \in \text{Ob}$, a class $\text{Arr}_{(A,B)}$ of *arrows*, where $f \in \text{Arr}_{(A,B)}$ is denoted by $f : A \rightarrow B$,
- for all objects $A, B, C \in \text{Ob}$, a binary operation (for composing arrows): $\circ : \text{Arr}_{(B,C)} \times \text{Arr}_{(A,B)} \rightarrow \text{Arr}_{(A,C)}$,
- for each object $A \in \text{Ob}$, an identity arrow $\text{id}_A : A \rightarrow A$,

such that the following conditions hold:

- *Associativity*.
 $\forall A, B, C, D \in \text{Ob}, \forall f : A \rightarrow B, g : B \rightarrow C, h : C \rightarrow D, (h \circ g) \circ f = h \circ (g \circ f).$
- *Identity*.
 $\forall A, B \in \text{Ob}, \forall f : A \rightarrow B, (f \circ \text{id}_A = f) \wedge (\text{id}_B \circ f = f).$

¹ Note that the more general term “class” (not to be confused with “class” in an OO context) and not “set” must be used in Def. 1 as, for example, the class of all sets is *not* a set (i.e., is a *proper* class) due to Cantor’s paradox (a set that contains its power set is “too big” to be a set).

To give an intuition for Def. 1 in an MDE context, imagine a category as a simple transformation system. Objects are models with some “black box” internal structure that is not revealed at this level, while arrows are transformations that allow a transition from the source model (object) to the target model (object). With this informal analogy of a transformation system for objects and arrows in mind, it makes sense to demand identity arrows for all objects (doing nothing always gets you the identity), and to demand a composition function on arrows (if two transformations can be applied one after the other, then this is obviously a way to get from the first to the last model in the chain).

In fact, even the laws are intuitive: as long as you apply transformations in the same sequence you get the same result, and composing identities with any other transformations does not make a difference.

As a final point, category theory tends to focus on *arrows* rather than objects as can be seen from the formulation of the conditions in Def. 1. This focus also fits to our analogy of a *transformation* system.

In the rest of this thesis, formal arguments based on categorical structures will be presented in a visual manner using *diagrams*. This is also referred to as *diagram chasing* [86] and is one of the advantages of category theory (at least for the visually inclined!).

As a trivial example, the conditions (using the same quantification as in Def. 1) for a category can be stated by simply demanding that the diagrams in Fig. 2.1 *commute*. In a commutative diagram, all possible paths of arrows between two objects are the same, i.e., in our transformation system analogy, the transformations obtained by composing different arrows between a source and target model behave exactly the same way (and are thus identical for all we care). Note that not all existing arrows and objects have to be depicted in a diagram – in Fig. 2.1::a,² for instance, the identity arrows are of course present but do not need to be depicted in the diagram. If only some parts of a diagram commute, this is denoted with a “=” symbol in the area enclosed by the commutative paths of arrows.

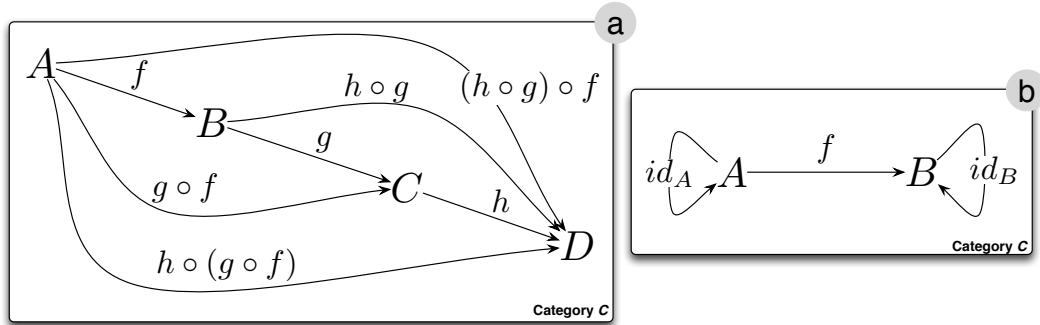


Figure 2.1: Commutative diagrams for associativity and identity conditions

To start with, we now introduce the category of sets³ and total functions, on which all other structures we shall treat are ultimately built on.

² Fig. X::B denotes diagram B (or in general the area/element marked by the label B) in Fig. X.

³ Subscripts, e.g., \circ_{Sets} instead of just \circ , are used in the following to explicitly denote a concrete category. If this is clear from the context, however, the subscripts are left off to improve readability.

Definition 2 (*Sets and Total Functions*).

Sets = $(\text{Ob}_{\text{Sets}}, \text{Arr}_{\text{Sets}}, \circ_{\text{Sets}}, \text{id}_{\text{Sets}})$ consists of:

- finite sets Ob_{Sets} ,
- total functions Arr_{Sets} ,
- for $A, B, C \in \text{Ob}_{\text{Sets}}$, $f : A \rightarrow B$, $g : B \rightarrow C$,
 $(g \circ_{\text{Sets}} f) : A \rightarrow C$ is defined as $\forall x \in A, (g \circ_{\text{Sets}} f)(x) := g(f(x))$,
- for $A \in \text{Ob}_{\text{Sets}}$,
 $\text{id}_A : A \rightarrow A$ is defined as $\forall x \in A, \text{id}_A(x) := x$.

Fact 1 (*Sets is a Category*).

Proof. (Sketch) Associativity and identity conditions follow directly from Def. 2 and well-known properties of total functions and function composition. The reader is referred to, e.g., Example 1.1.3 in [83] for a detailed proof. \square

Example 1 (*A diagram in Sets*).

Figure 2.2 depicts an example in **Sets**, with (a) showing the actual contents of the sets E and V , and the element mapping of the functions s and t . The same diagram is repeated in Fig. 2.2:(b) representing the sets now as atomic objects and the functions as arrows. A main goal of category theory is to present results based on (b) without “looking into” the internal structure of objects and arrows as in (a), allowing for a natural generalization of results to other categories.

The example presents a GOTO command with its arguments (cf. `v0.cls` in Fig. 1.2). As indicated by the chosen names (E for edges, V for vertices), the functions s (source) and t (target) assign to each edge source and target vertices (nodes), yielding a graph. Note that the elements in V are single graph nodes and not attribute values or sets of attribute values. There are, therefore, three separate ARGUMENT nodes representing the three separate arguments of a GOTO command, ignoring their actual coordinate values (attribution will be handled in Sect. 4.1). This specification of a graph-like structure is formalized next.

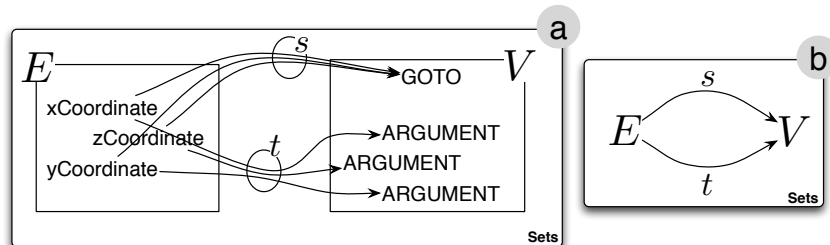


Figure 2.2: A diagram with objects and arrows in **Sets**

Remark 1 (*Totality vs. Partiality*).

In the rest of this thesis, *total* functions are simply referred to as functions, i.e., a function is total unless it is explicitly stated as being *partial*.

Definition 3 (*Graphs and Graph Morphisms*).

A *graph* $G = (V, E, s, t)$ consists of a set V of nodes (vertices), a set E of edges, and two functions $s, t : E \rightarrow V$ that assign each edge a source and target node, respectively.

Given graphs $G = (V, E, s, t)$, $G' = (V', E', s', t')$, a *graph morphism* $f : G \rightarrow G'$, $f = (f_V, f_E)$ consists of two functions $f_V : V \rightarrow V'$ and $f_E : E \rightarrow E'$ such that Fig. 2.3::a commutes for s, s' ($f_V \circ s = s' \circ f_E$) and t, t' ($f_V \circ t = t' \circ f_E$).

Graphs = $(\text{Ob}_{\text{Graphs}}, \text{Arr}_{\text{Graphs}}, \circ_{\text{Graphs}}, \text{id}_{\text{Graphs}})$ consists of:

- graphs $\text{Ob}_{\text{Graphs}}$,
- graph morphisms $\text{Arr}_{\text{Graphs}}$,
- for $G, G', G'' \in \text{Ob}_{\text{Graphs}}$, $f = (f_V, f_E) : G \rightarrow G'$, $g = (g_V, g_E) : G' \rightarrow G''$, $g \circ_{\text{Graphs}} f : G \rightarrow G''$ is defined as $g \circ_{\text{Graphs}} f := (g_V \circ_{\text{Sets}} f_V, g_E \circ_{\text{Sets}} f_E)$,
- for $G = (V, E, s, t) \in \text{Ob}_{\text{Graphs}}$, $\text{id}_G : G \rightarrow G$ is defined as $\text{id}_G := (\text{id}_V, \text{id}_E)$.

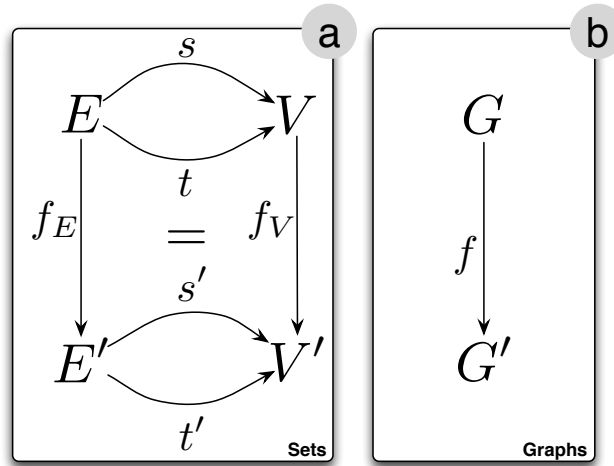


Figure 2.3: Graph morphisms are arrows that preserve the structure in graphs

Fact 2 (**Graphs** is a Category).

Proof. For arrows f, g in **Graphs**, we have to show that $(g \circ_{\text{Graphs}} f)$ is again an arrow in **Graphs**. To demonstrate how this can be shown via diagram chasing, consider Fig. 2.4 below, showing $(g \circ_{\text{Graphs}} f)$ as a diagram in **Sets** in (a), and as a diagram in **Graphs** in (b). Note that in each diagram, it is clear which composition (\circ) is meant and so we can leave off the respective subscript.

According to Def. 3, $(g \circ f)$ is an arrow in **Graphs** if it is structure preserving, i.e., if the arrows $s'' \circ g_E \circ f_E$ and $g_V \circ f_V \circ s$ commute. Such paths can be traced (chased) in the diagram (it helps to read \circ as *after*).

For s, s'' this can be reasoned as follows (t, t'' analogously):

$$s'' \circ g_E \circ f_E \xRightarrow{\text{Def. 2}} g_V \circ s' \circ f_E \xRightarrow{\text{Def. 2}} g_V \circ f_V \circ s.$$

Associativity and identity conditions follow directly from the respective conditions in **Sets**. \square

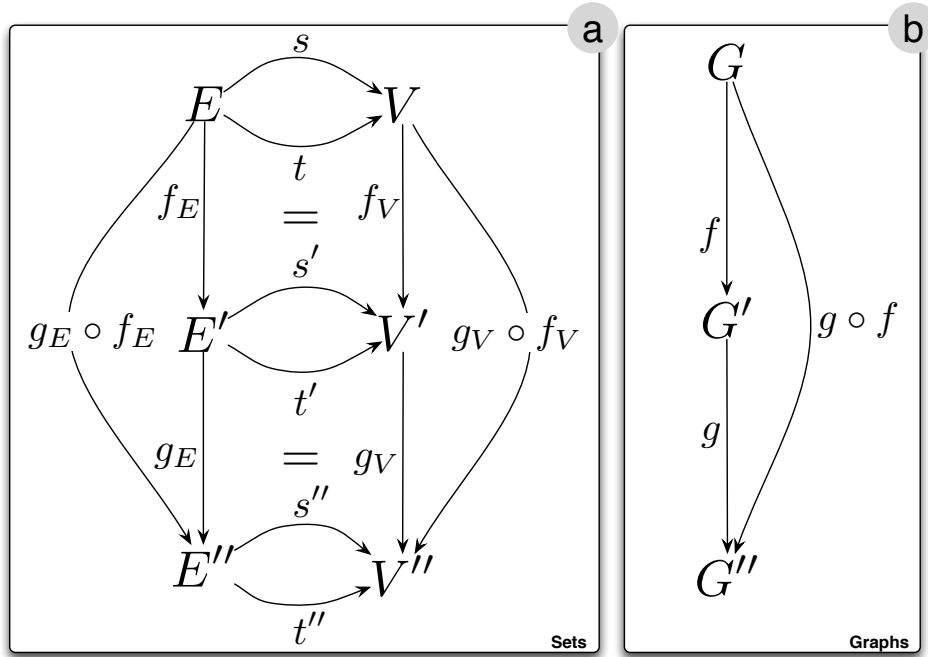


Figure 2.4: Composition of graph morphisms is sound

Example 2 (*GOTO pattern matching in Graphs*).

Figure 2.5 depicts the GOTO structure in `v0.mpf` (cf. Fig. 1.2) in diagram (a) as the graph G' . Note that for presentation purposes, a compact representation for graphs is now used, displaying edges as connections between nodes (source and target functions are implicitly given), and not as an extra set as in Fig. 2.2. There are three GOTOs in `v0.mpf`, the first with all coordinate values, the second only changing the value of its y coordinate, and the third only its x coordinate value. This is represented in G' by three GOTO nodes chained together by `next` and `prev` edges and connected to their arguments appropriately. The graph G contains a single GOTO with all its arguments. The graph morphism $m : G \rightarrow G'$ maps the nodes and edges in G to nodes and edges in G' . The *image* $m(G)$ of G in G' is denoted by giving the nodes identifiers (e.g., $1:\text{GOTO}$ in G is mapped by m to $1:\text{GOTO}$ in G'). Edges do not need identifiers in this case as the mapping for edges is implicitly fixed by their “labels”⁴ (edges can only be mapped to edges with the same label) and the mapping of source and target nodes. To improve readability, the rest of G' is greyed out. The graphs G , G' and graph morphism m are repeated in Fig. 2.5:(b) as a diagram in **Graphs**.

To provide an intuition for arrows in **Graphs** and the structure they preserve, G can be regarded as a *pattern* that is to be located in the host graph G' . The process of determining the image $m(G)$ is, therefore, called *pattern matching*, and the arrow m is called a *match morphism*. The image $m(G)$ of G in G' is called a *match* for G in G' . The condition for structure preservation required for graph morphisms in Def. 3 can now be understood as representing the expectation that the pattern G be matched faithfully in G' . G can be viewed as a set of conditions or constraints of the form: “there must exist a GOTO, which must be connected to an ARGUMENT via the x edge, and ...”.

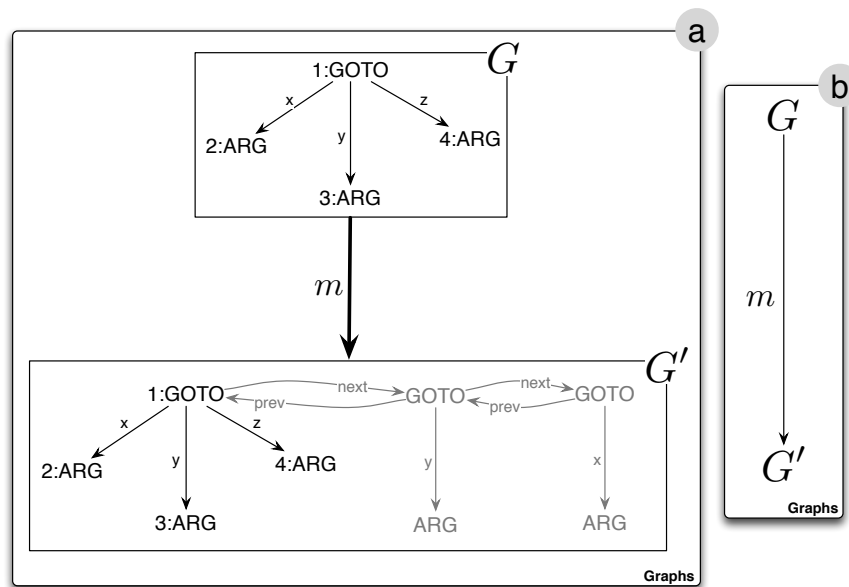


Figure 2.5: Matching a certain GOTO in a chain of GOTOs

⁴ Note that this is an informal explanation to motivate and provide an intuition for *types*. Labels will not be formalized as we shall use types instead.

A match for a given pattern must be determined by assigning all nodes and edges in the pattern to nodes and edges in the host graph, *fulfilling* all these structural constraints in the process. In practice, pattern matching is a basic and crucial task for graph transformation, and is performed by a *pattern matching engine*, an “arrow finder” that is able to construct an arrow $m : G \rightarrow G'$ given the pattern G and host graph G' .

The nodes and edges in G and G' (Fig. 2.5) are *labelled*, e.g., with either GOTO or ARGUMENT, and we have informally imposed the restriction that nodes and edges can only be mapped to nodes and edges with the same label. Formally, however, there is nothing enforcing this yet. Furthermore, it is not yet “forbidden” to connect, e.g., ARGUMENTs with next and prev edges. These additional conditions can be formalized by introducing the concept of type conformance for graphs.

Definition 4 (*Typed Graphs and Typed Graph Morphisms*).

A *type graph* is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$.

A *typed graph* is a pair $\hat{G} = (G, \text{type})$ of a graph G together with a graph morphism $\text{type} : G \rightarrow TG$.

Given typed graphs $\hat{G} = (G, \text{type})$ and $\hat{G}' = (G', \text{type}')$, a *typed graph morphism* $f : \hat{G} \rightarrow \hat{G}'$ is a graph morphism $f : G \rightarrow G'$ such that Fig. 2.6::a commutes, i.e., $\text{type}' \circ f = \text{type}$ (f is type preserving).

$\mathbf{TGraphs} = (\text{Ob}_{\mathbf{TGraphs}}, \text{Arr}_{\mathbf{TGraphs}}, \circ_{\mathbf{TGraphs}}, \text{id}_{\mathbf{TGraphs}})$ consists of:

- typed graphs $\text{Ob}_{\mathbf{TGraphs}}$,
- typed graph morphisms $\text{Arr}_{\mathbf{TGraphs}}$,
- $\circ_{\mathbf{TGraphs}} := \circ_{\mathbf{Graphs}}$,
- $\text{id}_{\mathbf{TGraphs}} := \text{id}_{\mathbf{Graphs}}$.

$\mathcal{L}(TG) := \{G \mid \exists \text{type} : G \rightarrow TG\}$ denotes the set of all graphs of type TG .

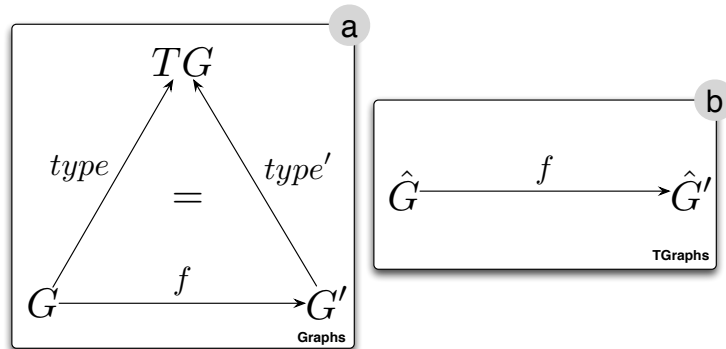


Figure 2.6: Typed graph morphisms are type preserving graph morphisms

Fact 3 (**TGraphs** is a Category).

Proof. (Sketch) Soundness of composition in **TGraphs** is shown by checking via simple diagram chasing (in this case a combination of commutative triangles results in a commutative triangle) that $f \circ_{\mathbf{Graphs}} g$ is indeed a typed graph morphism for typed graph morphisms f and g .

Associativity and identity conditions are inherited from **Graphs**. The reader is referred to, e.g., Example A.6 in [28] for a detailed proof. \square

Example 3 (*Type preserving GOTO pattern matching*).

Using objects and arrows in **TGraphs**, we can now make typing constraints in our example explicit. Figure 2.7 depicts the same match $m : G \rightarrow G'$ in diagrams (a) and (b) in **Graphs**, and now as a type preserving **TGraphs** arrow in (c). For presentation purposes, Fig. 2.7::(a) fixes the element mapping for m , $type$, $type'$ implicitly, by demanding that nodes and edges be mapped to nodes and edges of the same type. Identifiers are used where necessary to ensure that this mapping can be uniquely determined from the diagram.

Using our intuition based on pattern matching, type preservation ensures that *type constraints* in the pattern G , e.g., “there must exist a node of *type* GOTO, which must be ...” are fulfilled by the match m . Furthermore, the existence of type morphisms for every typed graph ensures that the graphs structurally conform to their typed graph. For example, it is now explicitly enforced that only GOTOs can be connected via *next* and *prev* edges.

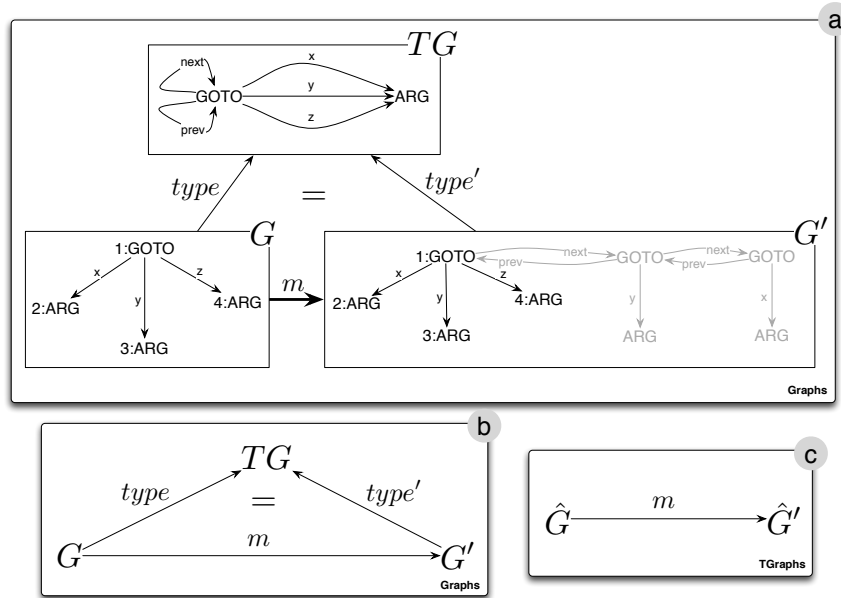


Figure 2.7: Matching a certain GOTO in a chain of GOTOs in a type preserving manner

An intuition for the pushout construction in **TGraphs** is that the graphs R and G are placed beside each other and then glued by merging all elements with the same pre-image in L , which are in this sense identical. The universal property ensures that there is no redundancy in this gluing; as soon as elements are glued together unnecessarily (without having the same pre-image in L) then the result cannot be the pushout as there would exist a G'' in which these elements are *not* glued together but there would exist no $x : G' \rightarrow G''$, violating the universal property. Conversely, *all* elements with the same pre-image in L *must* be merged (minimal gluing) as (1) must commute. This ensures, therefore, that “enough” elements are merged when gluing R and G .

Fact 4 (*Pushouts exist in Sets, Graphs, and TGraphs*).

In **Sets** the pushout object over arrows $r : L \rightarrow R, m : L \rightarrow G$ can be constructed as the quotient $R \uplus G |_{\equiv}$, where \uplus is the disjoint union of sets and \equiv is the smallest equivalence relation such that $\forall a \in L, (r(a), m(a)) \in \equiv$. The arrows r', m' are defined by: $\forall b \in G, r'(b) = [b]$ and $\forall c \in R, m'(c) = [c]$, where $[x]$ is the equivalence class for x with respect to \equiv .

In **Graphs** and **TGraphs** pushouts can be constructed componentwise for nodes and edges in **Sets**.

Proof. The interested reader is referred to Fact 2.17 in [28]. □

Example 4 (*Pushout in Sets*).

Figure 2.9 depicts an example, where the pushout of two arrows $r : L \rightarrow R, m : L \rightarrow G$ is constructed in **Sets**. The arrows r and m map elements with the same label to each other, e.g., $r(1_L) = 1_R, m(2_L) = 2_G, \dots$. The disjoint union $R \uplus G$ is depicted in the diagram already indicating which elements are in the same equivalence class with respect to \equiv with a dashed border. The pushout object G' is constructed in a final step by building the quotient set $R \uplus G |_{\equiv}$. This is done by collapsing all equivalence classes to a single representative. For $\{1_R, 1_G\}$ and $\{2_R, 2_G\}$, the chosen representative is depicted as 1, 2, respectively.

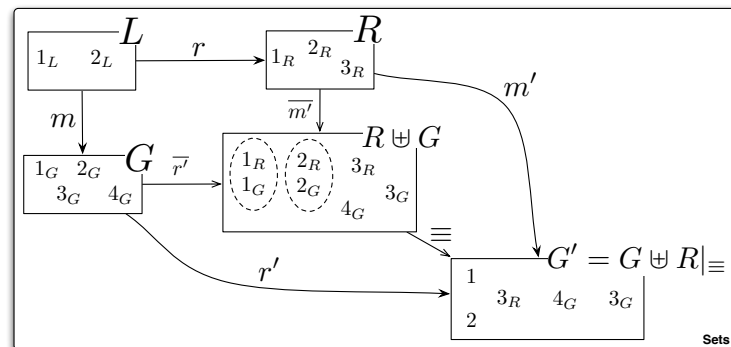


Figure 2.9: Constructing a pushout in Sets

As already indicated by the choice of letters in Fig. 2.8, i.e., L for *left-hand side* (the graph before the transformation), R for *right-hand side* (the graph after the transformation), a rule $r : L \rightarrow R$ is simply a morphism describing which elements are to be newly added to a match $m(L)$ of the left-hand side of the rule in the host graph G to produce the resulting graph G' . To simplify later definitions and formal results of this thesis, we shall restrict rule application to match morphisms $m : L \rightarrow G$ that are *injective* on nodes and edges. This is formalized in the following:

Definition 6 (*The Class \mathcal{M} of Monomorphisms*).

Given a category \mathbf{C} , an arrow $p : P \rightarrow G$ is a *monomorphism* if

$[\forall x, y : L \rightarrow P \in \text{Arr}_{\mathbf{C}}, p \circ x = p \circ y \text{ (Fig. 2.10 commutes)}] \Rightarrow [x = y]$.

The class of all monomorphisms in \mathbf{C} is denoted by \mathcal{M} .

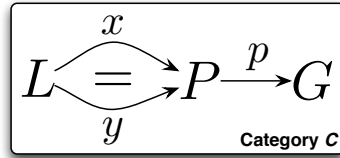


Figure 2.10: Diagram used to characterize monomorphisms

Definition 7 (*Initial Object*).

Given a category \mathbf{C} , an object \emptyset is called *initial* if, for each object A , there exists a unique arrow $\alpha_{\emptyset} : \emptyset \rightarrow A$.

Fact 5 (*Monomorphisms and Initial Object in $\mathbf{TGraphs}$*).

Graph morphisms that are injective on nodes and edges form the class \mathcal{M} of monomorphisms in $\mathbf{TGraphs}$ [28].

The initial object in $\mathbf{TGraphs}$ is the empty graph [28].

Definition 8 (*Rules, Graph Grammars, and Derivations*).

A *rule* is a typed graph morphism $r : L \rightarrow R \in \mathcal{M}$, where TG is a type graph and $L, R \in \mathcal{L}(TG)$.

A *graph grammar* is a pair $GG = (TG, \mathcal{R})$ of a type graph TG and a finite set \mathcal{R} of rules.

A *direct derivation* $G \xRightarrow{r@m, m'} G'$ is given by a pushout in **TGraphs** (cf. Fig. 2.8), where $m \in \mathcal{M}$ (and thus $m' \in \mathcal{M}$ according to Fact 2.17 in [28]).

The notation $G \xRightarrow{r@m, m'} G'$ is used to indicate that the rule r is applied to G at match m to yield G' with comatch m' , and is also abbreviated to $G \xRightarrow{r@m} G'$, or simply $G \xRightarrow{r} G'$, when the exact comatch/match is not relevant for the current discussion.

A *derivation* $G \xRightarrow{*} G'$ of length $n \geq 0$ in a graph grammar GG is a sequence of n direct derivations $G \xRightarrow{r_1} G_1 \xRightarrow{r_2} \dots \xRightarrow{r_n} G'$, with $r_1, r_2, \dots, r_n \in \mathcal{R}$.

In case of $n = 0$, we have $G' = G$.

$\mathcal{L}(GG, G_\emptyset) := \{G \in \mathcal{L}(TG) \mid G_\emptyset \xRightarrow{*} G\}$ denotes the language generated by a graph grammar GG , where G_\emptyset denotes the typed start graph.

$\mathcal{L}(GG) := \mathcal{L}(GG, \emptyset)$, where \emptyset is the empty typed graph.

Example 5 (*Rules for constructing chains of GOTOs*). —————

Figure 2.11 depicts a set of rules for constructing a chain of connected GOTOs with ARGs. Each of the five rules is depicted to the right as a diagram in **TGraphs**, and to the left in a *compact notation* for rules.

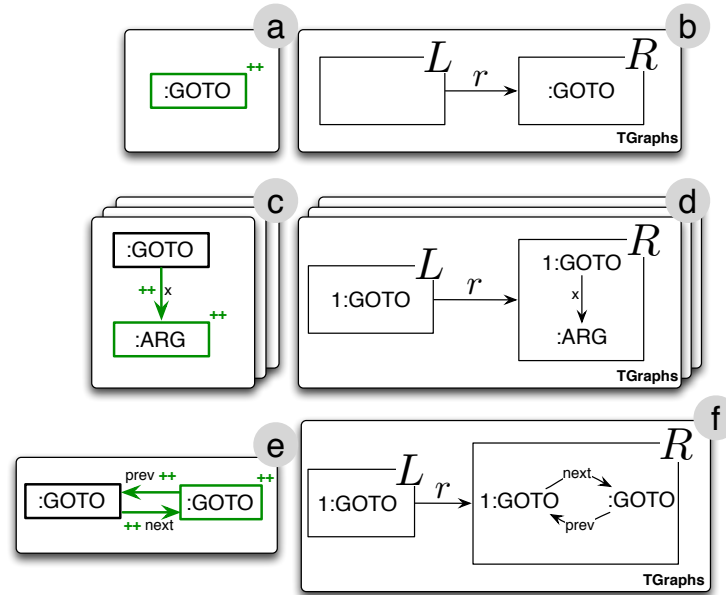


Figure 2.11: A set of rules for constructing a chain of GOTO commands

In the compact notation, L and R of the rule are merged together in a single diagram, depicting elements in $R \setminus L$ in green and additionally with a “++” markup to distinguish them from elements in L . In this manner it is always possible to determine the corresponding formal diagram in **TGraphs** from the

compact notation. Recall that for presentation purposes, typing is indicated implicitly in both notations (formal and compact) via the convention that a node with label :GOTO (or optionally with an id e.g., 1:GOTO) is of type GOTO. Similarly, an edge with label next is of type next (cf. Fig. 2.7).

Depending on the context and which syntax better supports the current discussion, both the compact and formal notation will be used interchangeably in the rest of this thesis.

The first rule (a)/(b) creates a new GOTO from the empty start graph, the second, third and fourth rules (c)/(d) create and connect new ARGs to an existing GOTO via an x, y, and z edge, respectively. The fifth rule (e)/(f) extends a chain of GOTOs by a new GOTO connected with both prev and next edges.

To demonstrate how a rule is applied by constructing a pushout, Fig. 2.12 depicts a direct derivation $G \xrightarrow{r} G'$ of Rule (e)/(f). First of all, a match m of L in the host graph G is determined. This match chooses the area of application of the rule in the host graph, i.e., in this case 1:GOTO and not 2:GOTO, which would also be a possible match for the rule. To apply the rule at the match $m(L)$, the pushout object G' is constructed by first building the disjoint union of all nodes and edges in G and R , and then merging the node 1:GOTO from G and 1:GOTO from R as they share the same pre-image 1:GOTO in L . This results in the graph G' and morphisms m' (referred to as a *co-match*) and r' . PO is used to indicate that a certain diagram is to be interpreted as a pushout diagram.

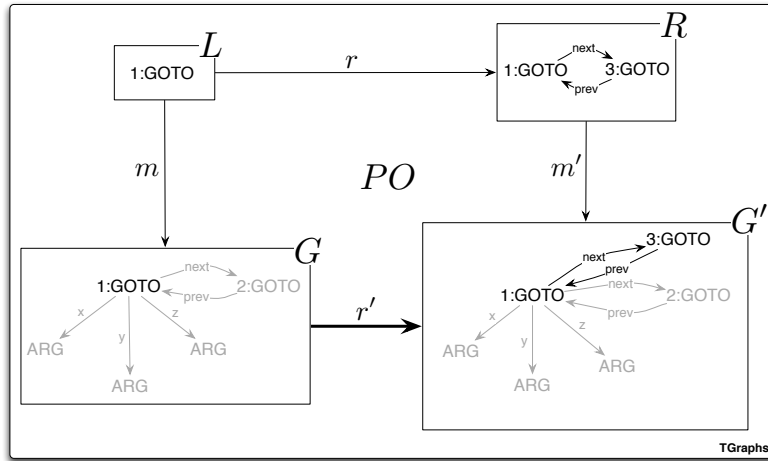


Figure 2.12: Applying a rule to extend a chain of GOTOs

Given a type graph such as TG in Fig. 2.7, it is sometimes necessary to further restrict the set of typed graphs $\mathcal{L}(\text{TG})$ by demanding that a set of conditions \mathcal{C} be fulfilled. In our case, for instance, it makes sense to forbid “trees” of GOTO commands, as it is unclear how to generate code from such models. The graph G' in Fig. 2.12 should thus violate a corresponding *condition* in \mathcal{C} .

Similarly, it is often useful to be able to control rule application by expressing further conditions that a match for the rule must satisfy. For example, the direct derivation in Fig. 2.12, which leads to the invalid graph G' , should not have been possible in the first place, i.e., an *application condition* should have blocked

this particular direct derivation of the rule as it leads to an invalid graph. Both these kinds of conditions are formalized in a uniform manner with the following definitions.

Definition 9 (Conditions).

Let \mathbf{C} be a category with a class \mathcal{M} of monomorphisms and initial object \emptyset .

A *condition* c over an object L is a pair $(x, \forall c_i)$, where $x : L \rightarrow P$ and $c_i : P \rightarrow C_i$, with $i \in I$, for some index set I . L is referred to as the *context*, P the *premise*, and $\{C_i \mid i \in I\}$ the *conclusions* of the condition.

An arrow $m : L \rightarrow G$ satisfies a condition $c = (x, \forall c_i)$, denoted by $m \models c$, if for all $p : P \rightarrow G \in \mathcal{M}$ such that (1) in Fig. 2.13 commutes, there exists $i \in I, q_i : C_i \rightarrow G$ such that (2) in Fig. 2.13 also commutes.

A condition $c = (x, \forall c_i)$ is *trivial* if $\exists i \in I, c_i : P \rightarrow P$, i.e., the condition is always fulfilled as c_i is the identity on P .

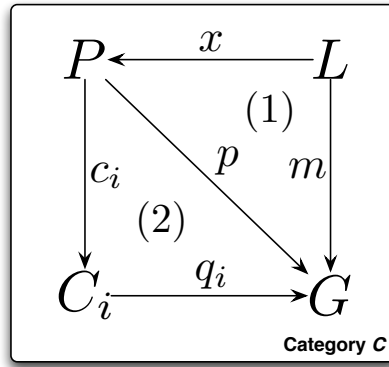


Figure 2.13: Satisfaction of conditions

Definition 10 (Constraints and Negative Constraints).

A *constraint* is a condition over the initial object \emptyset , i.e., $x : \emptyset \rightarrow P$.

An object G satisfies a constraint $c = (x, \forall c_i)$, denoted by $G \models c$, if $m_\emptyset : \emptyset \rightarrow G \models c$.

A *negative constraint* is a constraint with an empty set I of conclusions.

Negative constraints are “negative” as they are violated as soon as there exists a match for the premise P (there can be no conclusion as I is empty). In this sense, the premise P is forbidden (negated) in G . Negative constraints can, therefore, be denoted compactly by a single graph N instead of $(x : \emptyset \rightarrow N, I = \{\})$.

Definition 11 (*Application Conditions and Negative Application Conditions*).

A *precondition* c over a rule $r : L \rightarrow R$ is a condition over L .

A *postcondition* c over a rule $r : L \rightarrow R$ is a condition over R .

An *application precondition* c for a rule $r : L \rightarrow R$ is a precondition over r .

An *application postcondition* c for a rule $r : L \rightarrow R$ is a postcondition over r .

An *application condition* c for a rule $r : L \rightarrow R$ is an application precondition or postcondition for r .

A *Negative Application Condition* (NAC) is an application condition with an empty set I of conclusions.

A direct derivation $G \xRightarrow{r@m} G'$ satisfies an application precondition c for a rule r , denoted by $G \xRightarrow{r@m} G' \models c$, if $m \models c$.

A direct derivation $G \xRightarrow{r@m_\lambda m'} G'$ satisfies an application postcondition c for a rule r , denoted by $G \xRightarrow{r@m_\lambda m'} G' \models c$, if $m' \models c$.

Analogously to *negative* constraints, NACs are “negative” as they are violated as soon as there exists a match for the premise, which is thus “negated” in this sense. NACs can, therefore, be compactly denoted by a single morphism $n : L \rightarrow N$.

Definition 12 (*Graph Languages*).

Given a set \mathcal{C} of conditions, let $X \models \mathcal{C}$ be a compact notation for $\forall c \in \mathcal{C}, X \models c$.

A derivation $G \xRightarrow{*} G'$ satisfies a set of application conditions \mathcal{C} , denoted by $G \xRightarrow{*} G' \models \mathcal{C}$, if every direct derivation $G_i \xRightarrow{r} G_j$ in the derivation sequence satisfies the subset of application conditions $\mathcal{C}_r \subseteq \mathcal{C}$ for the rule r .

$\mathcal{L}(\text{TG}, \mathcal{C}_{\text{TG}}) := \{G \in \mathcal{L}(\text{TG}) \mid G \models \mathcal{C}_{\text{TG}}\}$ denotes the set of all *valid* graphs (with respect to the set \mathcal{C}_{TG} of constraints) of type TG .

$\mathcal{L}(\text{GG}, \mathcal{C}_{\text{GG}}) := \{G \in \mathcal{L}(\text{GG}) \mid G_\emptyset \xRightarrow{*} G \models \mathcal{C}_{\text{GG}}\}$ denotes the set of all graphs that can be generated by *valid* derivations (with respect to \mathcal{C}_{GG}) in GG .

$\mathcal{L}(\text{GG}, \mathcal{C}_{\text{GG}}, \mathcal{C}_{\text{TG}}) := \mathcal{L}(\text{TG}, \mathcal{C}_{\text{TG}}) \cap \mathcal{L}(\text{GG}, \mathcal{C}_{\text{GG}})$ denotes the set of all *valid* graphs that can be generated by *valid* derivations.

Example 6 (*Some conditions for our GOTO type graph and rules*). —————

Figure 2.14 depicts conditions for GOTO graphs and a GOTO rule. Diagram ① is a constraint requiring that every GOTO command must have a coordinate value. This makes sense in our case study as a GOTO command without changing any coordinates would mean that the previous values of all coordinates are retained, i.e., the machine is not moved at all. G and G' in Fig. 2.5 both satisfy this constraint, while G and G' in Fig. 2.12 do not.

Diagram (b) is a *negative* constraint, as the index set I is empty. This means that a graph G violates the constraint as soon as a $p : P \rightarrow G$ exists, i.e., P can be seen as a forbidden structure that should never occur in valid graphs. P enforces a strict linear chain of GOTOs connected via next edges. An analogous negative constraint can be formulated for GOTOs connected via prev edges.

Diagrams (c) and (d) depict the same application condition for $r : L \rightarrow R$ in Fig. 2.11::(e)/(f). Diagram (c) uses a compact notation for representing *negative* application conditions ($I = \{ \}$), by merging P in the same diagram, distinguishing elements in $P \setminus L$ by crossing them out. In this case, the condition states that a new GOTO can only be appended to a GOTO that does not yet have a successor (via next). Note that the next reference does not need to be crossed out explicitly in the compact notation as this is clearly the case (all incident edges of crossed-out nodes are automatically crossed out to ensure that L is a graph). Together with an analogous condition for prev edges, the language generated by the graph grammar can be thus restrained to linear GOTO chains. Diagram (d) (in **TGraphs**) repeats the same condition in the usual detailed formal syntax.

Diagram (e) depicts another application condition for the same rule, this time ensuring that a GOTO chain is only extended if the matched GOTO already has at least one argument. In the context of the GOTO graph grammar in Fig. 2.11, this application condition would ensure that a coordinate rule (Fig. 2.11::(c)) must be applied before a new GOTO can be created to extend the chain at that position.

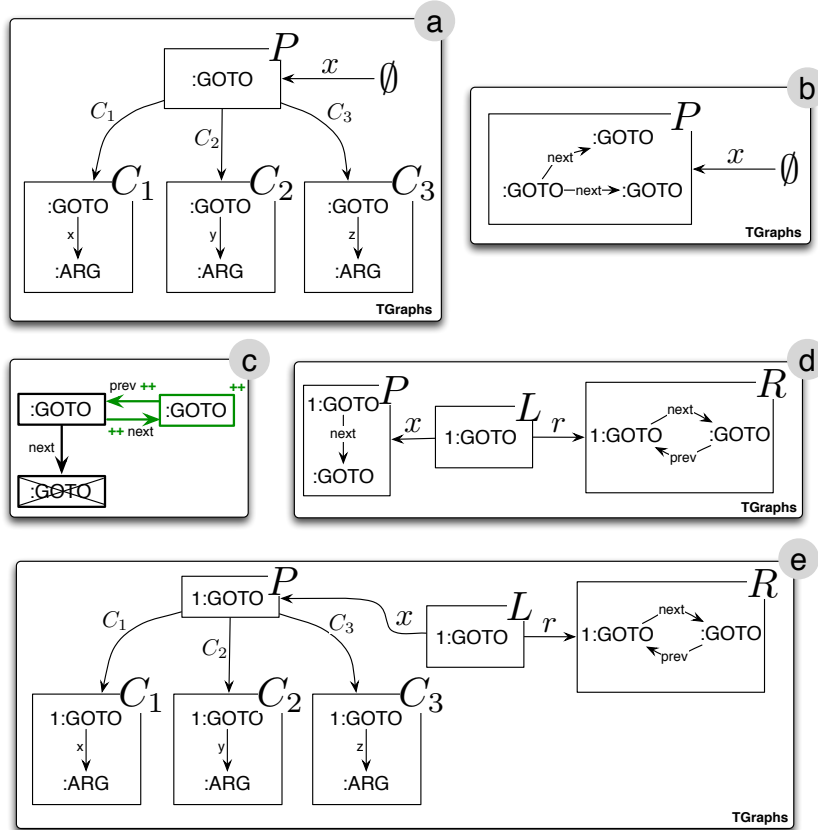


Figure 2.14: Conditions for GOTO graphs and rules

2.1.3 Overview of Practical MDE Concepts

We can now map a first set of basic MDE concepts to objects in our algebraic formalization. This represents *one possible* interpretation of these concepts, i.e., the same concepts can also be formalized e.g., with set theory or logic resulting in a different mapping and formal understanding. In the following, each term is defined informally in an MDE setting according to [12, 15, 97], before mapping it to the framework of algebraic graph transformations. As a concluding summary, Table 2.1 gives an overview of the resulting interpretation.

MODEL: A model of a system is an abstraction thereof that is particularly suitable for performing a certain task. Abstraction typically involves a simplification and reduction of details that are not relevant for the task driving the abstraction process.

Figure 2.15 depicts 6 models from the case study, including a CNC program ①, which already abstracts from machine code, and a G0T0 graph ②, which abstracts from the CNC program in a way that is suitable for synchronization with a corresponding CLS file.

↪ In the formal framework, a *model* corresponds to an *object* in **TGraphs**, i.e., a *typed graph*.

ABSTRACTION LEVELS AND “REPRESENTS” RELATION: In an MDE context where everything can be viewed as a model (of something else), creating models of models of models induces *abstraction* levels, where models on higher levels of abstraction *represent* models on lower levels of abstraction. This means that, with respect to the task driving the abstraction process, the more abstract model captures all *relevant* aspects of the less abstract model and is a valid (and hopefully more suitable!) substitute for performing this particular task. Figure 2.15 organizes 6 models in two abstraction levels on the horizontal axis ⑦: the abstraction level for models related to CNC, and the abstraction level for models related to our G0T0 language. As the G0T0 language is an abstraction of CNC,⁵ the models on this higher level of abstraction *represent* models on the abstraction of CNC, for instance, the G0T0 graph ② represents the CNC program ①.

↪ In the formal framework, the “represents” relation between models G' and G corresponds to a derivation $G \xRightarrow{*} G'$, via rules⁶ that change, remove, or add structure as necessary.⁷

META-LEVELS AND “CONFORMS TO” RELATION: In Fig. 2.15, the vertical axis ⑧ represents the *meta-levels* M1, M2 and M3. Models on a higher meta-level *define* the modelling language used to specify models on lower meta-levels. The latter are then said to *conform to* the former.

⁵ Details such as line numbers, comments, and other commands (e.g., to change the feedrate of the machine) are omitted.

⁶ Recall, however, that we have only defined monotonic (creating) rules

⁷ Unfortunately, this is a very rough approximation and does not truly capture the intuitive meaning of the “represents” relation.

Models on M1 are, for example, the CNC program file `v0.mpf` ① and the GOTO graph ② thereof. Models on M2 define the modelling language used to specify models on M1 and are thus referred to as *metamodels* to emphasize this. In Fig. 2.15, CNC programs such as ① conform to a definition of the CNC language ③. Similarly, GOTO models conform to a metamodel that defines the GOTO language ④. In turn, ③ is written in German, i.e., conforms to ⑤ that defines the German language in German, and is thus self conforming. The GOTO metamodel ④ conforms to a *meta-metamodel* ⑥ that defines a language for specifying metamodels. A concrete example of such a language (used in this thesis) is *Ecore* [90], which defines basic concepts for metamodeling such as *classes*, *references*, and *attributes*, as well as relationships between concepts such as *inheritance* via the *eSuperTypes* reference between *EClasses*, and *typing* via the *eType* reference between *ETypedElement* and *EClassifier*. Note that only a small excerpt of *Ecore* is depicted in ⑥ and that exact details are not particularly relevant for this thesis. Analogously to ⑤, *Ecore* is self-defining and self-conforming, i.e., *EClass* is an *EClass*. Models on M3 typically have this characteristic to “finalize” the conformance hierarchy.

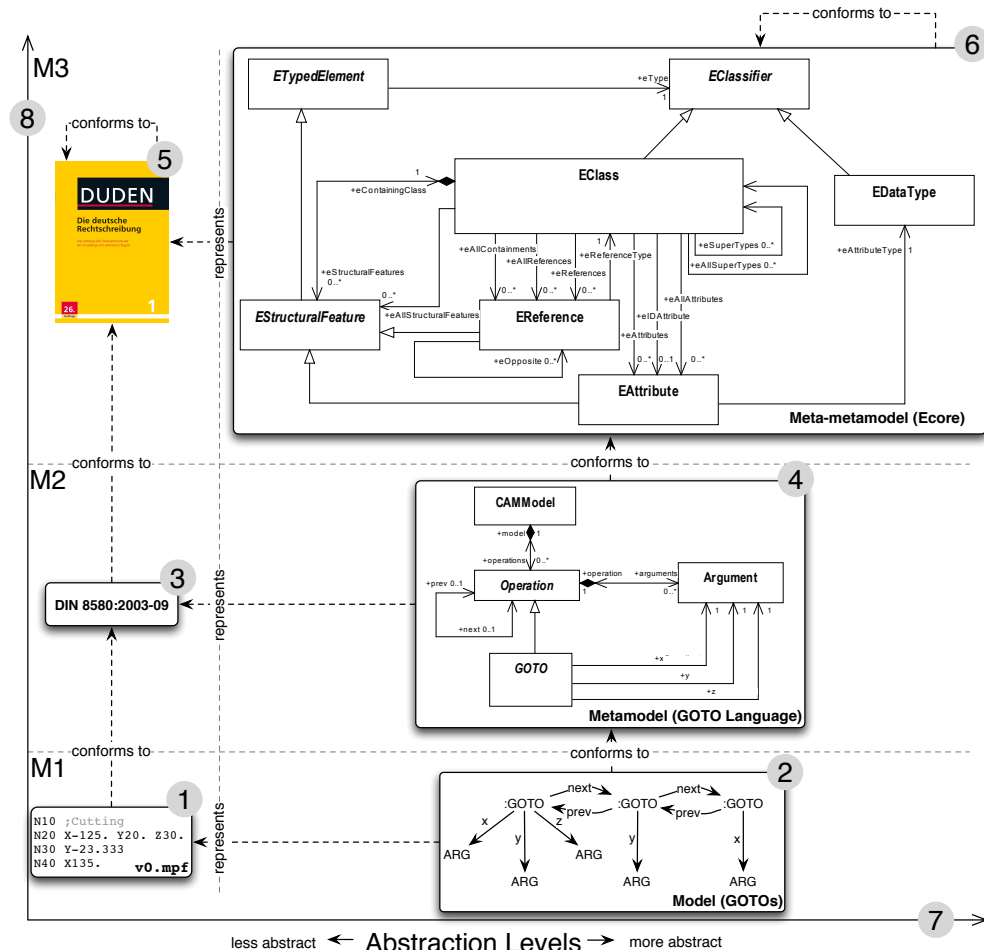


Figure 2.15: Meta-levels vs. abstraction levels

↪ In the formal framework, metamodels are typed graphs (models) that are also *type* graphs, while meta-metamodels⁸ are type graphs that happen to be their own type graph as well (metamodels). The “conforms to” relation, therefore, corresponds to the graph morphism *type* that exists between a typed graph and its type graph.

As Ecore supports further concepts from UML class diagrams such as inheritance, abstract classes, attributes and multiplicities, the complete mapping to the formal framework is a bit more involved.⁹ In this thesis, inheritance hierarchies and abstract classes are flattened according to [28], while attribution is formalized in Chap. 4, Sect. 4.1.

Multiplicities in metamodels are an example for constraints that are used to additionally restrain the language of valid models (that conform to the metamodel). Further constraints for metamodels can be specified with a constraint language such as OCL.

To be more precise, therefore, metamodels correspond to a type graph *and* a set of conditions that valid typed graphs must satisfy. Note, however, that although the form of conditions introduced in Def. 9 for this thesis covers multiplicities, only a small subset of OCL can be expressed.

MODEL TRANSFORMATION: A model transformation Δ takes a source model G as input and produces a target model G' as output. Model transformations are classified in detail in [21, 76] along various dimensions. Two basic dimensions are: (1) is the source metamodel different from the target metamodel (an *exogenous* transformation) or the same (an *endogenous* transformation)?, and (2) is the source model manipulated in the process (a *destructive* transformation)?

↪ In the formal framework, a model transformation Δ from G to G' corresponds to a derivation $G \xRightarrow{*} G'$ in a given graph grammar.

This means that endogenous transformations are naturally represented, while exogenous transformations can be simulated by regarding two metamodels as a single type graph and demanding via suitable conditions that the source model must conform to the source type graph and the target model to the target type graph. Similarly, derivations are in general destructive, but can be restrained to be non-destructive on the source graph.

⁸ The example implies the amusing conjecture that *Ecore* is an abstraction of the German language!

⁹ A further mismatch is that typing morphisms cannot directly express the conformance of meta-models to meta meta-models as e.g., *references* in meta-models are instances of a class in the meta meta-model.

MODEL SPACE: Given a set of model transformations and a starting model, a *model space* or *state space* (not defined formally in this thesis) can be created by constructing all possible derivations. A small excerpt of the (infinite) model space induced by the rules in Fig. 2.11 is depicted in Fig. 2.16.

↪ In the formal framework, a model space corresponds¹⁰ to the language $\mathcal{L}(\text{GG}, \mathcal{C}_{\text{GG}})$ induced by a graph grammar GG with conditions \mathcal{C}_{GG} . This model space can be restricted to valid models $\mathcal{L}(\text{GG}, \mathcal{C}_{\text{GG}}, \mathcal{C}_{\text{TG}})$.

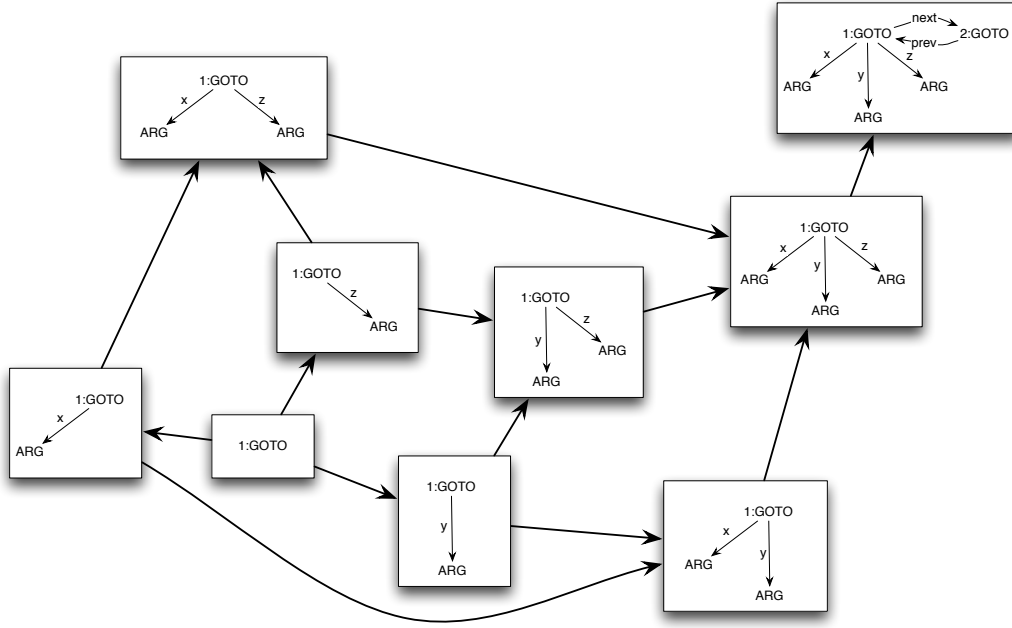


Figure 2.16: Excerpt of the model space for the GOTO language

¹⁰ Note, however, that only the language elements (models) are of interest for a *language*, while also the transitions (model transformations) between language elements are relevant for *state spaces*.

MDE Concept	Formalization
Model	Typed graph
Abstraction level and “represents” relation.	Derivation performing abstraction from one graph to another.
Meta-level	Depth of sequence of <i>type</i> graph morphisms.
“conforms to” relation between model and metamodel	Graph morphism <i>type</i> from a typed graph to its type graph.
Metamodel	Type graph TG and a set of conditions \mathcal{C}_{TG} over TG.
Meta-metamodel	Type graph (usually its own type graph as well) and a set of conditions.
Model Transformation	Derivation
Model Space	In general, the language $\mathcal{L}(GG, \mathcal{C}_{GG}, \mathcal{C}_{TG})$ of valid graphs induced by a graph grammar GG with application conditions \mathcal{C}_{GG} and constraints \mathcal{C}_{TG} over the type graph TG.

Table 2.1: Basic MDE concepts in the framework of algebraic graph transformations

2.2 MODEL SYNCHRONIZATION WITH TRIPLE GRAPH GRAMMARS

To extend our formalization to cover BX in general and *model synchronization* in particular, we take again a constructive, (graph) grammar-based approach. The central idea is to establish a means for specifying a *consistency relation* between two graph languages. It should then be possible to automatically derive *synchronizers* from the consistency relation, which are able to propagate changes to a graph in one of the languages, to changes to a corresponding graph in the other language.

In a grammar-based approach, such a consistency relation is specified as a pair of *coupled* graph grammars, each describing one graph language (let us refer to these languages as *source language* and *target language* as from now on). In this manner, a source graph is *consistent* if there exists a target graph, so that the pair of source and target graphs can be constructed with a derivation of coupled rules in the pair of source and target graph grammars. To formalize this *coupling*, the connection between consistent pairs of source and target graphs is materialized as a *correspondence* graph in its own right. This correspondence graph is often referred to generally as a *witness structure* for the consistency of the source and target graphs it connects. We now introduce a category of triples and triple morphisms to provide a suitable context for lifting Def. 8 to triples.

2.2.1 Consistency Specification with Triple Graph Grammars

In the following, we denote *triple graphs* with single letters, e.g., G , which consist of typed graphs with a subscript S (Source), C (Correspondence), or T (Target) indicating the language or *domain* the graph belongs to, e.g., G_S, G_C, G_T .

Definition 13 (*Triple Graphs and Triple Morphisms*).

A *triple graph* $G = G_S \xleftarrow{\sigma_G} G_C \xrightarrow{\tau_G} G_T$ consists of graphs G_S, G_C and G_T , and graph morphisms $\sigma_G : G_C \rightarrow G_S, \tau_G : G_C \rightarrow G_T$.

A *triple morphism* $f = (f_S, f_C, f_T) : G \rightarrow G', G' = G'_S \xleftarrow{\sigma_{G'}} G'_C \xrightarrow{\tau_{G'}} G'_T$ is a triple of graph morphisms $f_S : G_S \rightarrow G'_S, f_C : G_C \rightarrow G'_C, f_T : G_T \rightarrow G'_T$, such that Fig. 2.17::Ⓐ commutes.

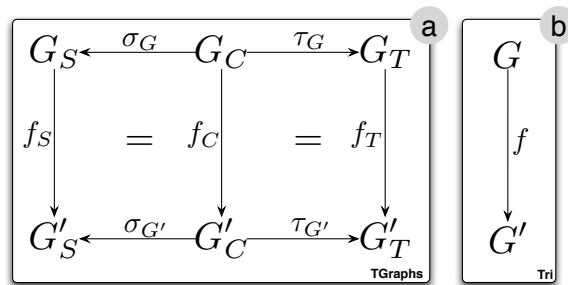


Figure 2.17: Triple morphisms preserve the triple structure in triple graphs

Definition 14 (*Typed Triple Graphs and Typed Triple Morphisms*).

Let a triple graph $TG = TG_S \xleftarrow{\sigma_{TG}} TG_C \xrightarrow{\tau_{TG}} TG_T$ be called *type triple graph*.

A *typed triple graph* is a pair (G, type) of a triple graph G and triple morphism $\text{type} : G \rightarrow TG$.

Analogously to Def. 4, $\mathcal{L}(TG)$ denotes the set of all triple graphs of type TG .

Given $(G, \text{type}), (G', \text{type}') \in \mathcal{L}(TG)$, a *typed triple morphism* $g : G \rightarrow G'$ is a triple morphism such that $\text{type} = \text{type}' \circ g$.

$\mathbf{Tri} = (\text{Ob}_{\mathbf{Tri}}, \text{Arr}_{\mathbf{Tri}}, \circ_{\mathbf{Tri}}, \text{id}_{\mathbf{Tri}})$ consists of:

- typed triple graphs $\text{Ob}_{\mathbf{Tri}}$,
- typed triple morphisms $\text{Arr}_{\mathbf{Tri}}$,
- for $G, G', G'' \in \text{Ob}_{\mathbf{Tri}}$, $f = (f_S, f_C, f_T) : G \rightarrow G'$,
 $g = (g_S, g_C, g_T) : G' \rightarrow G''$, $g \circ_{\mathbf{Tri}} f : G \rightarrow G''$ is defined as:
 $g \circ_{\mathbf{Tri}} f := (g_S \circ_{\mathbf{TGraphs}} f_S, g_C \circ_{\mathbf{TGraphs}} f_C, g_T \circ_{\mathbf{TGraphs}} f_T)$,
- for $G = TG_S \xleftarrow{\sigma_G} TG_C \xrightarrow{\tau_G} TG_T \in \text{Ob}_{\mathbf{Tri}}$,
 $\text{id}_G : G \rightarrow G$ is defined as $\text{id}_G := (\text{id}_{G_S}, \text{id}_{G_C}, \text{id}_{G_T})$.

Fact 6 (*\mathbf{Tri} is a Category with Pushouts, Initial Object and Set \mathcal{M} of Monomorphisms*).

\mathbf{Tri} with typed triple graphs as objects and typed triple morphisms as arrows forms a category with pushouts constructed componentwise in $\mathbf{TGraphs}$, the empty triple $\emptyset_S \xleftarrow{\sigma_\emptyset} \emptyset_C \xrightarrow{\tau_\emptyset} \emptyset_T$ as initial object, and the set \mathcal{M} of injective typed triple morphisms as monomorphisms.

Proof. The interested reader can refer to Fact 4.18 in [28] for a detailed proof. Soundness of composition is shown via basic diagram chasing (combination of commutative squares), typing arguments are analogous to Fact 3. Associativity and identity conditions are inherited componentwise from $\mathbf{TGraphs}$. \square

Example 7 (*A triple language of coupled GOTO and simulator models*).

Figure 2.18 depicts a type triple graph $TG = TG_S \xleftarrow{\sigma_{TG}} TG_C \xrightarrow{\tau_{TG}} TG_T$, and a triple graph $G = G_S \xleftarrow{\sigma_G} G_C \xrightarrow{\tau_G} G_T$, typed over TG , as diagram (a) in $\mathbf{TGraphs}$, and (b) in \mathbf{Tri} . The target language TG_T is our by now well-known GOTO abstraction of MPF, with G_T showing how GOTOs are connected and that arguments do not need to be repeated if they remain unchanged.

The source language TG_S introduces a new and equally simple abstraction of CLS files, a *simulator language* focusing on the sequence of commands in a CLS file and in particular on PAINT commands, used for visualization in a simulator. As PAINT commands in CLS always refer to (i.e., visualize) the previous

command, this is represented explicitly in the abstraction with a direct paint link between Command and PAINT objects. For presentation purposes, we assume that a single PAINT command is always sufficient for the visualization.

The correspondence type graph TG_C defines correspondence types for connecting related Commands and GOTOs, as well as PAINTs and ARGs. For presentation purposes, the typed graph morphisms σ_G and τ_G in a triple graph G are depicted as grey dashed arrows to differentiate them visually from other arrows. Note that these arrows do *not* represent edges!

To make things interesting, we assume that the CLS programmer only wants to visualize movements of the machine *that involve the x coordinate*. This means that the simulator language abstracts from any movement in the YZ-plane.

This explains why the source graph G_S , representing the CLS file below it, does not have a PAINT directly after the second GOTO, which does not change the x coordinate. The first and last GOTO are visualized, however, as the movement involves the x coordinate in both cases.

In our correspondence graph, we want to: (1) connect Commands with their respective GOTOs, and (2) connect PAINTs with the x coordinate of the GOTO they visualize. G_S and G_C are, therefore, consistent according to this informal specification (depicted in Fig. 2.18 as a large double-headed arrow between the CLS and MPF file) as G_C can be constructed as stipulated. The next definition formalizes consistency specification using triple graph grammars.

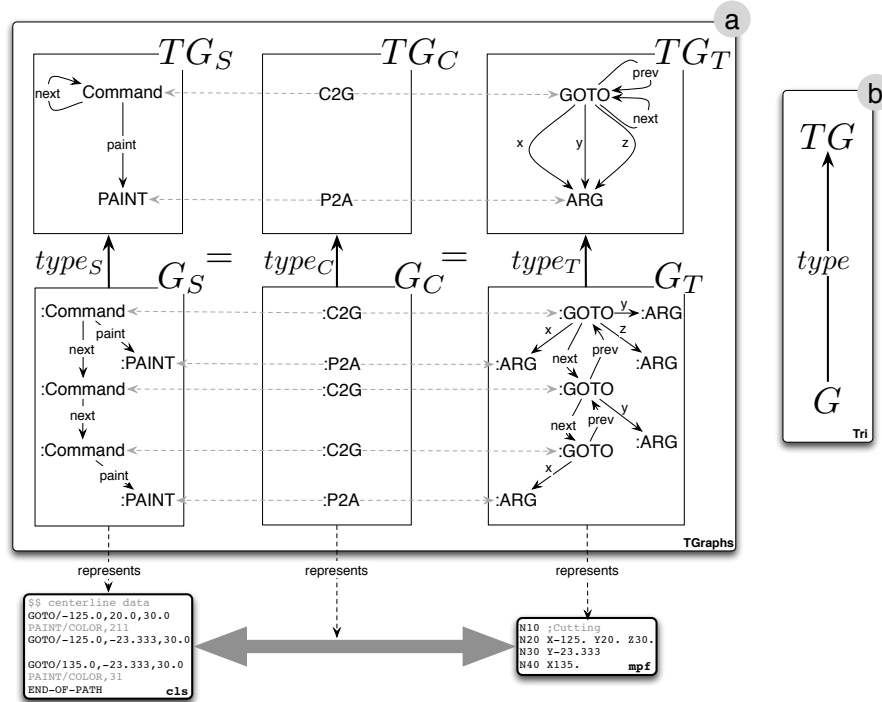


Figure 2.18: A type triple graph and typed triple graph

Definition 15 (*Triple Rules, Triple Graph Grammar*).

Let TG be a type triple graph, and $L, R \in \mathcal{L}(TG)$.

A typed triple morphism $r : L \rightarrow R$ is a *triple rule* if r_S, r_C , and r_T are rules.

A *triple graph grammar* $TGG = (TG, \mathcal{R})$ consists of a type triple graph TG and a finite set \mathcal{R} of triple rules.

Example 8 (*A triple graph grammar for consistent triples of GOTO and simulator models*).

Figure 2.19 depicts five rules, formalizing the consistency relation between simulator and GOTO models. The same compact syntax for rules introduced in Fig. 2.11 is used to represent triple rules. Remember, however, that the connections between correspondence and source/target elements represent graph morphisms, not edges.

Rule (a) creates a Command and a respective GOTO without requiring any context. Such rules are referred to as *axioms* as they can always be applied.

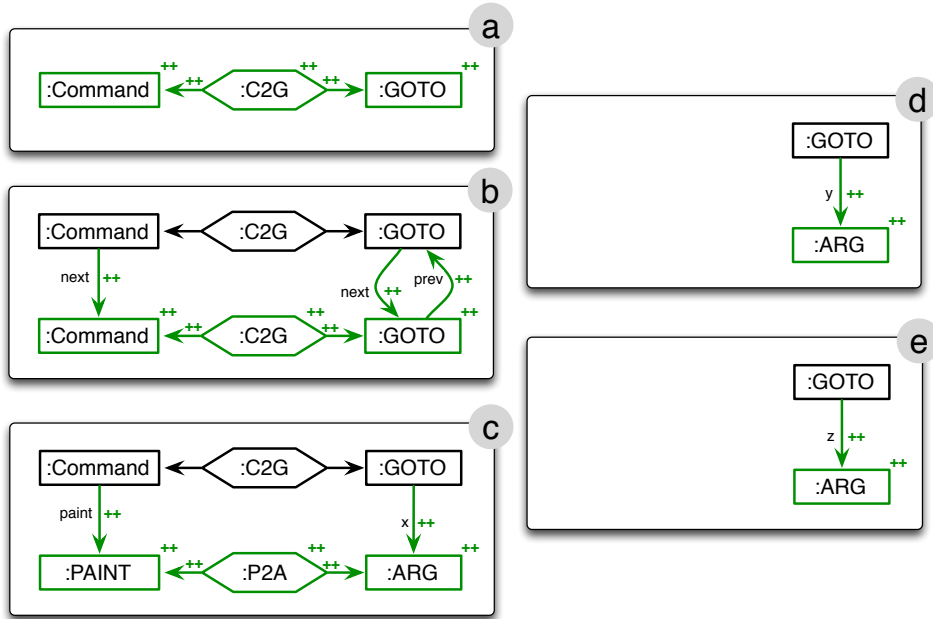


Figure 2.19: Triple rules for consistent GOTO and simulator models

Rule (b), on the other hand, prolongs an existing sequence of Commands and GOTOs simultaneously, requiring as context the previous Command and GOTO. This rule can, therefore, only be applied if the required context can be matched, i.e., there already exists a consistent Command-GOTO triple.

Similarly, Rule (c) creates a PAINT and a corresponding x coordinate ARG, appending them to an existing triple as well.

All rules until now have simultaneously extended both the source and the target graph. Rules (d), and (e), however, show that this is not always necessary. Both rules extend *only* the target graph when applied, without changing the

source and correspondence graphs in any way. Formally, such rules consist of L and R graph triples with trivial (empty) correspondence and target/source graphs. Rules with trivial source/target, but non-trivial correspondence and target/source graphs are also possible. In practice, as in our example here, such rules are used to formalize extensions of one language that have no effect on the consistency relation, and are thus referred to as *ignore rules*. In this case, adding y or z coordinates to GOTOs does not change the fact that a PAINT is (not) required for consistency. Note that this has important consequences: (1) there are multiple consistent source graphs for a single target graph, i.e., the correspondence between source and target graphs is *not* a bijection, and (2) not all changes to a graph have to be synchronized to restore consistency.

With this triple graph grammar, we can now check if G_S and G_T in Fig. 2.18 are consistent by checking if $G \in \mathcal{L}(\text{TGG})$. This is indeed the case as the following derivation of triple rules creates G (the exact match for each rule application is obvious from Fig. 2.18):

$$\textcircled{a} \rightarrow \textcircled{b} \rightarrow \textcircled{b} \rightarrow \textcircled{c} \rightarrow \textcircled{c} \rightarrow \textcircled{d} \rightarrow \textcircled{d} \rightarrow \textcircled{e}$$

2.2.2 Consistency Restoration via Delta Propagation

We are now able to define consistency relations using TGGs. Although it is already quite useful to be able to check if a given pair of source and target graphs are consistent, it is even *more* useful to be able to restore consistency given a triple and some changes made to one of the graphs.

These changes are referred to as a *delta*, meaning that consistency restoration (model synchronization) involves correct *delta propagation*. The following definitions formalize the concept of a delta in our algebraic framework.

Definition 16 (*Lifting Standard Operators to TGraphs and Tri*).

Let $G_S = (V, E, s, t)$, $G'_S = (V', E', s', t')$ be typed graphs, and $f_S : G_S \rightarrow G'_S$ a typed graph morphism.

The operators \in , $|\cdot|$, and (\cdot) are lifted from **Sets** to **TGraphs** as follows:

$$\begin{aligned} o \in G_S &\Leftrightarrow o \in V \vee o \in E \\ |G_S| &:= |V| + |E| \\ \forall o \in V_{G_S}, f_S(o) &:= f_{S_V}(o) \\ \forall o \in E_{G_S}, f_S(o) &:= f_{S_E}(o) \end{aligned}$$

Let $G = G_S \xleftarrow{\sigma_G} G_C \xrightarrow{\tau_G} G_T$, $G' = G'_S \xleftarrow{\sigma_{G'}} G'_C \xrightarrow{\tau_{G'}} G'_T$ be typed triple graphs, and $f : G \rightarrow G'$ a typed triple morphism.

The operators \in , $|\cdot|$, and (\cdot) are lifted from **TGraphs** to **Tri** as follows:

$$\begin{aligned} o \in G &\Leftrightarrow o \in G_S \vee o \in G_C \vee o \in G_T \\ |G| &:= |G_S| + |G_C| + |G_T| \\ \forall o \in G_S, f(o) &:= f_S(o) \\ \forall o \in G_C, f(o) &:= f_C(o) \\ \forall o \in G_T, f(o) &:= f_T(o) \end{aligned}$$

Definition 17 (*Source, Correspondence and Target Deltas*).

Let $G_S \xleftarrow{\sigma_G} G_C \xrightarrow{\tau_G} G_T$ and $G'_S \xleftarrow{\sigma_{G'}} G'_C \xrightarrow{\tau_{G'}} G'_T$ be typed triple graphs.

A *source delta* from G_S to G'_S , is a typed triple graph $\Delta S = G_S \xleftarrow{\delta_S^-} \Delta_S \xrightarrow{\delta_S^+} G'_S$.

$\Delta_S^- := \{o^- \in G_S \mid \nexists o \in \Delta_S, \delta_S^-(o) = o^-\}$ denotes the set of *deleted* elements.
 $\Delta_S^+ := \{o^+ \in G'_S \mid \nexists o \in \Delta_S, \delta_S^+(o) = o^+\}$ denotes the set of *created* elements.

A *correspondence delta* $\Delta C = G_C \xleftarrow{\delta_C^-} \Delta_C \xrightarrow{\delta_C^+} G'_C$ and
target delta $\Delta T = G_T \xleftarrow{\delta_T^-} \Delta_T \xrightarrow{\delta_T^+} G'_T$ are defined analogously.

A *delta* is a triple $\Delta S \xleftarrow{\delta_S} \Delta C \xrightarrow{\delta_T} \Delta T$ of source, correspondence, and target deltas, with typed triple morphisms $\delta S : \Delta C \rightarrow \Delta S$ and $\delta T : \Delta C \rightarrow \Delta T$.

Understanding the distinction between *consistent* and *inconsistent* deltas is important as it explains what can be reasonably expected from a model synchronization framework and what not. In practice, users usually have the natural but often unreasonable expectation to be able to propagate *any* delta. The following definition and example state that this is in general not possible.

Definition 18 (*Consistent Source Delta*).

Let $TGG = (TG, \mathcal{R})$ be a triple graph grammar, $G \in \mathcal{L}(TGG)$ a typed triple graph.

A source delta $\Delta S = G_S \xleftarrow{\delta_S^-} \Delta_S \xrightarrow{\delta_S^+} G'_S$ is *consistent* with respect to TGG if there exists a typed triple graph $G' = G'_S \xleftarrow{\sigma_{G'}} G'_C \xrightarrow{\tau_{G'}} G'_T \in \mathcal{L}(TGG)$.

Consistent correspondence and target deltas are defined analogously.

Example 9 (*GOTO and simulator deltas*).

Figure 2.20 depicts three deltas: Δ^1 (a) and Δ^2 (b) are both source deltas representing changes to G_S (Fig. 2.18), while Δ^3 (c) is a target delta for G_T (Fig. 2.18). Note that Δ^3 only deletes a single edge and does not create anything.

Interestingly, the source deltas result in isomorphic source graphs G'_S , but perform a different set of changes to achieve this. Δ^1 deletes 3:Command and its connected 5:PAINT, and re-creates a new 7:Command without a PAINT (handling edges as required for structure preservation). A new 6:PAINT is also created, connected now to 2:Command. Δ^2 is much less destructive, achieving the same effect as Δ^1 by only deleting and recreating a single paint edge.

Being able to distinguish between Δ^1 and Δ^2 characterizes a *delta-based* synchronization framework as opposed to a *state-based* setting, in which such deltas would be handled in exactly the same way.

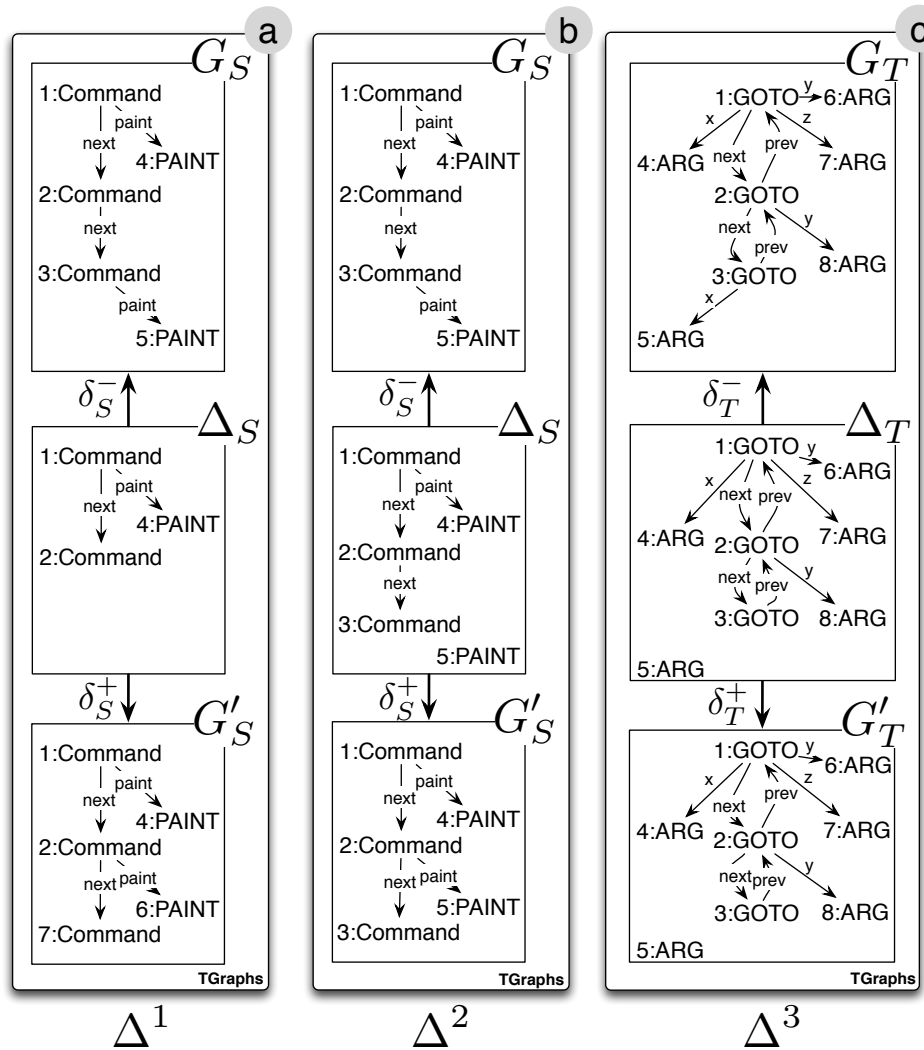


Figure 2.20: Deltas for GOTO and simulator models

Deltas Δ^1 and Δ^2 in Fig. 2.20 are both consistent as G'_S can be extended to a consistent triple $G' \in \mathcal{L}(\text{TGG})$. Delta Δ^3 , on the other hand, is *not* consistent as it is impossible to extend G'_T to a consistent triple in $\mathcal{L}(\text{TGG})$. This is because 5:ARG can only be created with Rules (c), (d), or (e) (Fig. 2.19). All these rules require, however, that 5:ARG be connected to a GOTO. As this is not the case, i.e., 5:ARG is isolated, a triple with G'_T as its target graph can never be created by a derivation in the TGG consisting of the rules depicted in Fig. 2.19. Partially propagating inconsistent deltas such as Δ^3 and still guaranteeing that synchronizers *perform as well as possible* is a current research challenge [94].

We are now ready to characterize and define *synchronizers* that are to be derived from a TGG by a TGG-based tool. Definition 19 and 20 are adapted from [71].

Definition 19 (TGG-Based Forward/Backward Synchronizers).

Let $\text{TGG} = (\text{TG}, \mathcal{R})$ be a triple graph grammar, $G \in \mathcal{L}(\text{TGG})$ a typed triple graph, and $\Delta S = G_S \xleftarrow{\delta_S^-} \Delta_S \xrightarrow{\delta_S^+} G'_S$ a consistent source delta.

A *forward synchronizer* for TGG is a partial function Sync_F that maps a consistent source delta ΔS to either a delta $\text{Sync}_F(\Delta S) = \Delta S \xleftarrow{\delta_S} \Delta_C \xrightarrow{\delta_T} \Delta T$ so that Fig. 2.21::(a) commutes, or to $\text{Sync}_F(\Delta S) = \perp$ (not defined for the source delta).

The resulting interconnected triple structure $\text{Sync}_F(\Delta S)$ (Fig. 2.21::(a)) can be interpreted as a diagram in **Tri** both *horizontally* as $\Delta S \xleftarrow{\delta_S} \Delta_C \xrightarrow{\delta_T} \Delta T$ (depicted in Fig. 2.21::(c)), and *vertically* as $G \xleftarrow{\delta^-} \Delta \xrightarrow{\delta^+} G'$ (depicted in Fig. 2.21::(b)).

A *backward synchronizer* Sync_B for TGG is defined analogously.

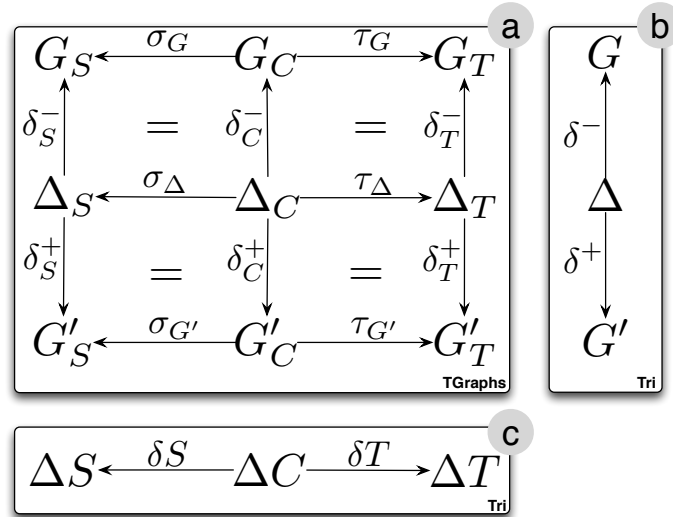


Figure 2.21: Resulting structure produced by TGG synchronizers

Example 10 (*Exemplary synchronization results*).

Figure 2.22 depicts a possible result of propagating Δ^1 (Fig. 2.20::**(a)**) using a TGG-based forward synchronizer. Note that although the resulting triple G' appears isomorphic to the result in Fig. 2.23, depicting possible results for propagating Δ^2 (Fig. 2.20::**(b)**), the synchronizer has applied a different target delta ΔT in each case. In practice, this has a substantial impact on efficiency and the perceived quality of the synchronizer.

Note that both results (Fig. 2.22 and Fig. 2.23) have both been carefully characterized as *possible* synchronization results. This is because the “expected” behaviour of synchronizers is not apriori clear and the corresponding target delta is seldom unique. Figure 2.24, for instance, depicts an alternate synchronization result for propagating Δ^2 (Fig. 2.20::**(b)**). The resulting triple G' is certainly consistent, but intuitively, this behaviour is not as “good” as that portrayed by Fig. 2.23. The following definitions present a set of properties to formalize this expected/optimal behaviour of TGG-based synchronizers.

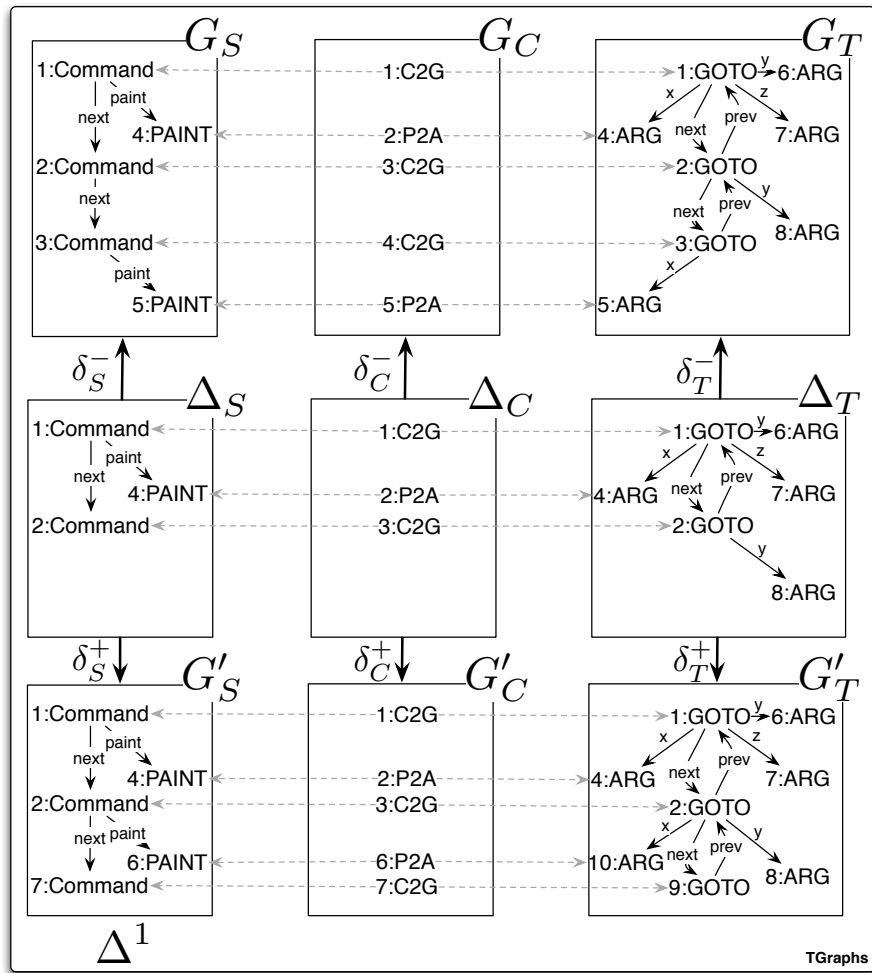


Figure 2.22: Synchronization results of propagating Δ^1

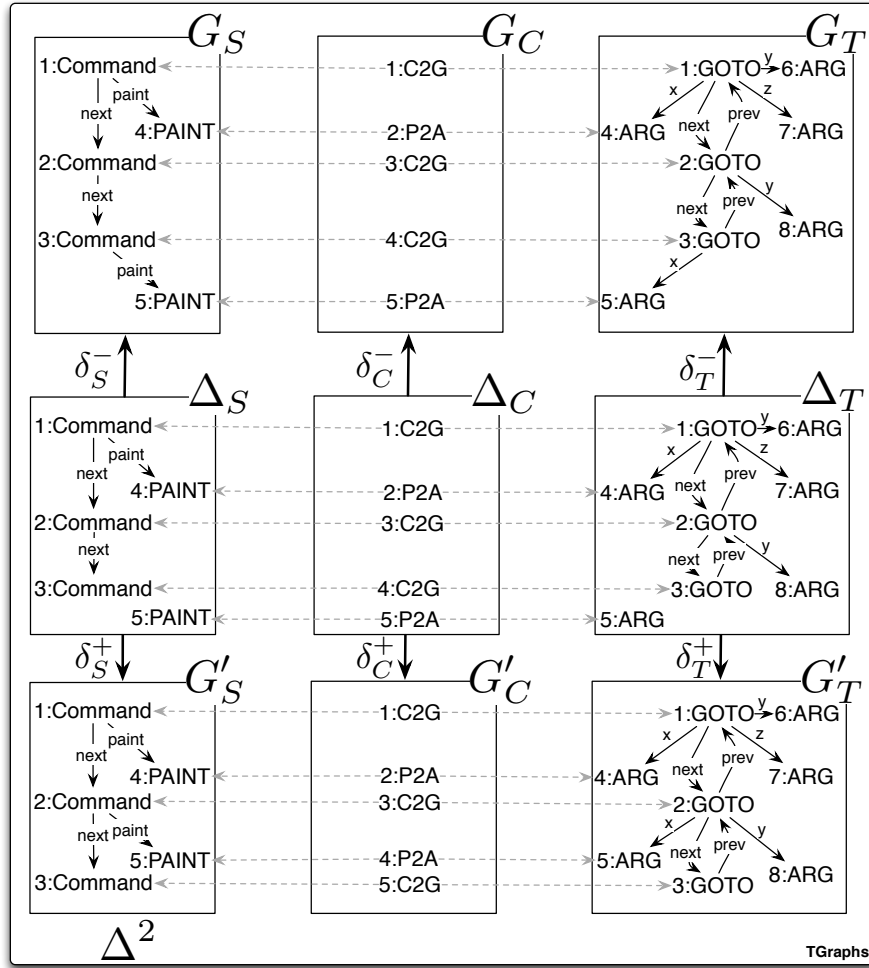
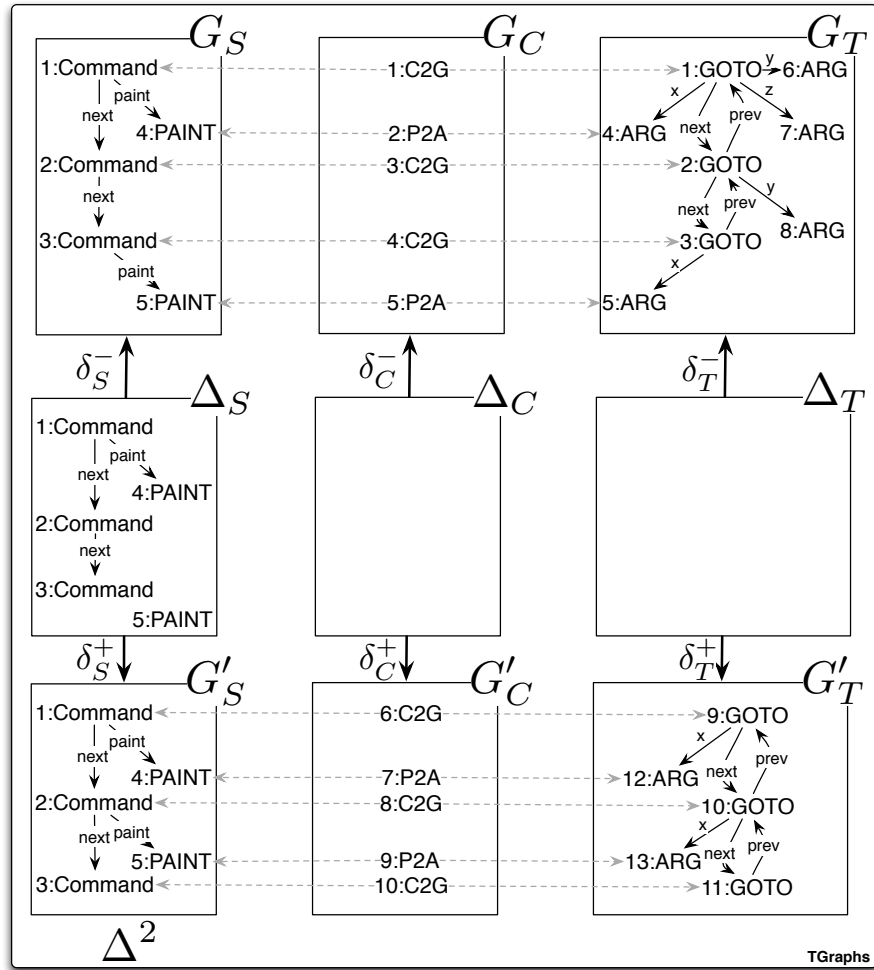


Figure 2.23: Synchronization results of propagating Δ^2

Figure 2.24: Alternate synchronization results of propagating Δ^2

Definition 20 (*Formal Properties of TGG-Based Synchronizers*).

Let $TGG = (TG, \mathcal{R})$ be a triple graph grammar, $G \in \mathcal{L}(TGG)$ a typed graph triple, and $\Delta S = G_S \xleftarrow{\delta_S^-} \Delta_S \xrightarrow{\delta_S^+} G'_S$ a consistent source delta.

The following properties are defined for a forward synchronizer Sync_F for TGG.

CORRECTNESS:

$$[\text{Sync}_F(\Delta S) = G \xleftarrow{\delta^-} \Delta \xrightarrow{\delta^+} G'] \Rightarrow [G' \in \mathcal{L}(TGG)].$$

COMPLETENESS:

Sync_F is *total*, i.e., defined for all consistent source deltas ΔS .

INCREMENTALITY:

$$[\text{Sync}_F(\Delta S) = G \xleftarrow{\delta^-} \Delta \xrightarrow{\delta^+} G'] \Rightarrow [\nexists G \xleftarrow{\hat{\delta}^-} \hat{\Delta} \xrightarrow{\hat{\delta}^+} G', \hat{\Delta} \sqsupset \Delta],$$

where \sqsupset is a suitable partial order on deltas.

In the rest of this thesis, *incrementality* will be defined using injective embeddings as a partial order on deltas:

$$\hat{\Delta} \sqsupset \Delta \Leftrightarrow \exists e : \Delta \rightarrow \hat{\Delta} \in \mathcal{M}$$

This means that delta $\hat{\Delta}$ preserves a superset of the elements already preserved by Δ , i.e., $\hat{\Delta}$ deletes and/or creates fewer elements than Δ .

This is clearly a very simple partial order and is not helpful when deltas are not comparable in this manner. A characterization of a more powerful (less partial) order on deltas is an open research question.

EFFICIENCY:

$$[\text{Sync}_F(\Delta S) = G \xleftarrow{\delta^-} \Delta \xrightarrow{\delta^+} G'] \Rightarrow [|\text{Sync}_F|(\mathbf{n}) \in \mathcal{O}(\mathbf{n}^k)]$$

where :

$$\mathbf{n} := |\Delta| := |\Delta_S^- \cup \Delta_S^+ \cup \Delta_C^- \cup \Delta_C^+ \cup \Delta_T^- \cup \Delta_T^+|$$

$$k := \max(\{|\mathcal{R}| \mid r : L \rightarrow R \in \mathcal{R}\})$$

$|\text{Sync}_F|$:= complexity of the algorithm used to implement Sync_F , given as a function of $|\Delta|$

$$\mathcal{O}(g(\mathbf{n})) := \{f(\mathbf{n}) \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0, 0 \leq f(\mathbf{n}) \leq c \cdot g(\mathbf{n})\}$$

Properties for backward synchronizers are defined analogously.

Example 11 (*Expected behaviour of synchronizers*). —————

The most basic property for any synchronizer is probably correctness. A synchronizer must guarantee that consistent deltas are propagated to a consistent result, ensuring that a chain of delta propagation can be realized and leads to a consistent result. All results depicted in Fig. 2.22, 2.23, and 2.24 are correct as the respective G' is always a member of $\mathcal{L}(TGG)$.

For efficiency reasons, TGG tools typically restrict the class of supported TGG specifications in different ways. For this reason, it is important to demand *completeness* for a well-defined subclass of TGGs. This means that for all TGGs that satisfy a set of conditions, a *complete* synchronizer must be able to propagate all possible consistent deltas. In practice, this is challenging to ensure and involves a careful compromise between restrictions and guaranteed efficiency.

To give an intuition for incrementality based on our example, the alternate synchronization result depicted in Fig. 2.24 is less incremental than the results in Fig. 2.23, as Δ_T in Fig. 2.24 deletes all elements and recreates only what is necessary to ensure consistency of the resulting triple G' . In fact, this result is not incremental at all and is what can be expected from a TGG state-based synchronizer, often referred to as a TGG batch transformation. The input triple and source delta are basically ignored, and G'_S is simply extended to a consistent triple using the TGG rules. The fact that it is impossible to recreate elements such as $6:ARG$, which do not have any correspondence in G_S , i.e., are created using *ignore rules* that do not extend the structure in one domain, is referred to as *information loss*. Incremental synchronizers strive to avoid information loss completely.

The final property *efficiency* demands that a synchronizer exhibit polynomial runtime in the size of the resulting delta. Note that it only makes sense to compare this property for synchronizers that already exhibit a comparable grade of incrementality. The definition generalizes to batch transformations in a sensible manner, but it is not “fair” to directly compare incremental and batch approaches with respect to efficiency.

Finally, efficiency implies that if the resulting delta is decoupled in size from the rest of the graph triple, an incremental synchronizer must work locally and exhibit the same runtime irrespective of the size of the complete graph triple being updated. This is a substantial technical challenge in practice.

2.2.3 Overview of Practical BX Concepts

To continue our interpretation of MDE terms in our formal framework, the following extends the mapping to cover basic terms in the BX community (cf. [22, 91]). As there are various formal frameworks, this is again only one possible interpretation of these concepts in an algebraic, TGG-based setting.

BX: The first point to understand when working with BX is that *bidirectionality* does not mean or imply *bijectivity*. A BX is simply a pair of transformations, one for a *forward* transformation of source to target graphs, and a *backward* transformation of target to source graphs. For the BX to be useful, however, the two transformations must be “compatible” with each other. This can be ensured in many different ways: (i) by deriving e.g., the backward transformation from the forward or vice-versa, (ii) by checking a set of compatibility properties of a given pair of forward and backward transformations, (iii) by providing a core set of atomic BXs and a means to compose them to larger BXs, or, finally, (iv) by deriving both forward and backward transformations from a single high-level specification (this is the strategy taken by TGG-approaches).

↪ In the presented TGG-based synchronization framework, a BX is given by a pair of forward and backward synchronizers $\text{Sync}_F, \text{Sync}_B$ derived from a TGG.

STATE-BASED VS. DELTA-BASED BX: BX frameworks can be grouped into state-based and delta-based approaches. This influences the set of properties that can be required from BXs. In a state-based setting, the only input available for a forward synchronizer is a consistent triple G , and a new state of the source graph G'_S . This means that a state-based synchronizer *cannot* distinguish between Deltas (a) and (b) depicted in Fig. 2.20, and cannot be expected to handle them differently.

↪ TGG-based synchronizers are clearly delta-based. Although this has the advantage of enabling more fine-grained synchronization, it can be problematic in a practical setting. It is often difficult and costly (inefficient) to calculate input deltas if the application scenario is inherently state-based.

INCREMENTAL UPDATE, DELTA PROPAGATION: Both these terms are used interchangeably to refer to the process of creating a new consistent target graph G'_T from a previous consistent pair G_S, G_T and a source delta ΔS . These terms reflect the fact that this process can be interpreted as *incrementally* updating G_T , as well as *propagating* ΔS to a ΔT .

↪ This duality of update and propagation is presented formally in Def. 19. The resulting structure after applying a synchronizer can be viewed vertically as an update, or horizontally as a delta extension/propagation. Although both representations in **Tri** are equivalent, depending on the current discussion, one representation might be more suitable than the other (cf. Def. 19 and 20).

INFORMATION LOSS: In general, both source and target languages in a practical synchronization scenario usually have both relevant and irrelevant elements with respect to the synchronization. Special cases include *bijective* BXs (no irrelevant elements) and *views* (only the source language contains irrelevant elements). In the latter case, the term view is used for the target graph as it is fully determined by the source graph. In these simplified cases, the transformation that produces the view is said to incur *information loss* as it is impossible to reproduce the source graph only from the view (i.e., without knowledge of the previous source graph).

↪ In a TGG-based framework, “irrelevant elements” of a language are formalized using *ignore rules* that do not create any elements of the source or target language such as Rules (d) and (e) in Fig. 2.19. This means that the backward synchronizer derived from a TGG with ignore rules for the target language can incur information loss (e.g., as depicted in Fig. 2.24) if it is not incremental enough.

MODEL SYNCHRONIZATION VS. INTEGRATION: Up until now, we have presented and discussed delta propagation under the assumption that *only one graph* is changed in a single synchronization step. In practice, however, both graphs are usually changed concurrently. This means that a synchronizer must be able to update a consistent graph G with *both* a source and target delta. To differentiate these tasks in this thesis, propagation of deltas from one domain to the other is referred to as *model synchronization*, while the propagation of deltas in both domains in a single step is referred to as *model integration*. As model integration involves conflict handling, e.g., via policies that decide what types of deltas are “dominant”, a different set of properties is required. This is out of scope for this thesis.

BX Concept	Formalization
Bidirectional Transformation (BX)	A pair of Sync_F and Sync_B derived from the same TGG.
State-based vs. Delta-based BX	Given $G_S \xleftarrow{\delta_S^-} \Delta_S \xrightarrow{\delta_S^+} G'_S$, state-based approaches only use G'_S .
Incremental Update, Delta Propagation	$G \xleftarrow{\delta^-} \Delta \xrightarrow{\delta^+} G', \Delta_S \xleftarrow{\delta_S^-} \Delta_C \xrightarrow{\delta_T^+} \Delta_T$ as defined in Def. 19.
Information Loss	Ignore rules in a TGG that do not create any new elements in the source or target domain.
Model Synchronization vs. Integration	TGG-based synchronizers can (currently) only handle deltas in one domain in a single step (synchronization and not integration).

Table 2.2: Interpretation of basic BX concepts in a TGG-based synchronization framework

2.2.4 Vertical and Horizontal Model Synchronization

Figure 2.25 depicts a schematic overview of a TGG-based synchronization landscape in an MDE context for our (still very much simplified) case study in the CME domain.

Two abstraction levels are shown on the vertical axis: *low* for the textual formats CLS and MPF, and *high* for the abstractions (models) chosen for the synchronization. The horizontal axis, on the other hand, spans the CLS and MPF domains.

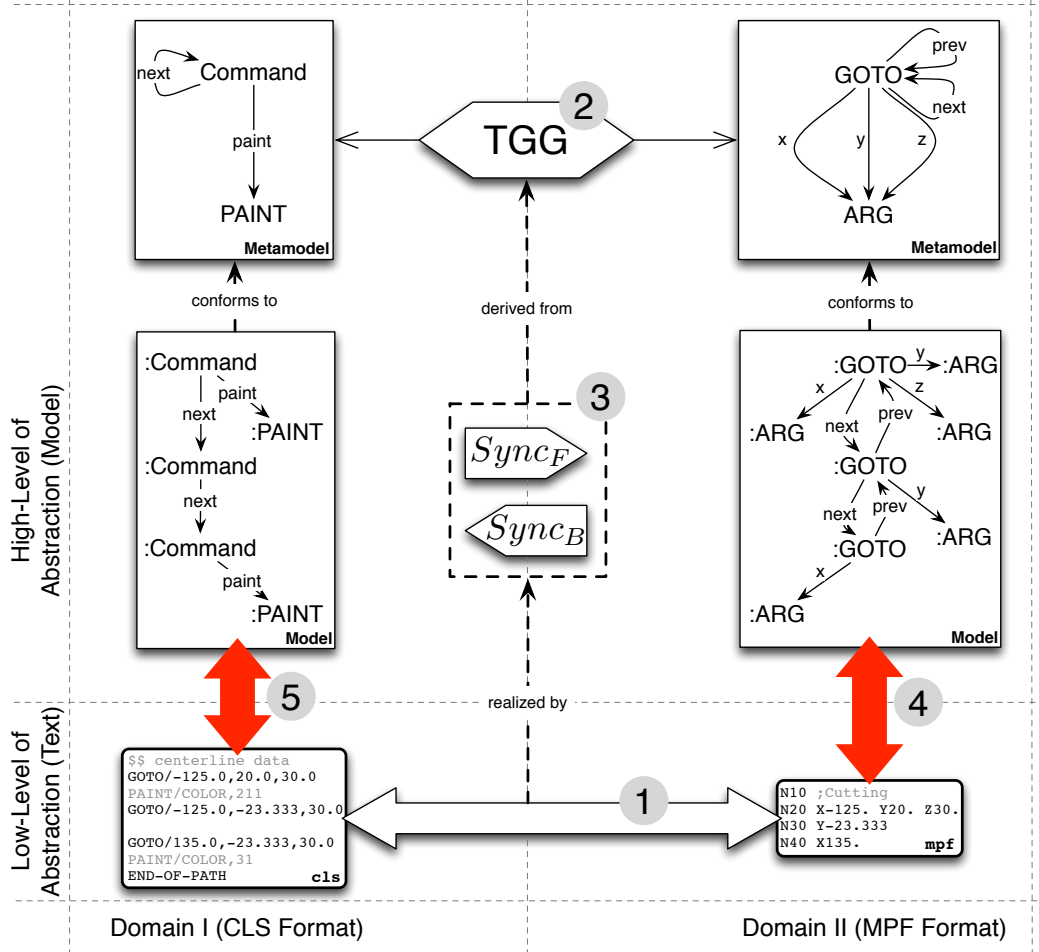


Figure 2.25: Horizontal and vertical model synchronization

In this chapter on fundamentals of TGG-based model synchronization, we have investigated with a simple example how to realize a synchronization of CLS and MPF files according to a certain notion of consistency ①. On a suitable level of abstraction, we were able to capture this notion of consistency with a high-level, declarative set of TGG rules ②. Current state-of-the-art TGG-based approaches are able to derive synchronizers ③ automatically from the TGG, which realize the synchronization in a consistent manner obeying the laws stipulated in Def. 20.

This type of TGG, which spans multiple domains but remains on the same level of abstraction is said to realize a *horizontal* synchronization. Almost all work on

TGGs up to now, has been focused on horizontal TGGs, assuming that suitable abstractions for the synchronization already exist.

In practice, however, such abstractions have to be established first, typically starting from textual artefacts such as program code or XML files arranged in a certain folder structure. In Fig. 2.25, these transformations are depicted by the red (dark) arrows ④ and ⑤. In a complete synchronization landscape, these transformations must also be bidirectional and incremental. In fact, the requirements are identical to those for horizontal synchronization. Such transformations that traverse different abstraction levels but typically remain in one domain are referred to as *vertical* transformations.

The question to be answered in the next chapter is how such transformations can also be realized with *vertical* TGGs providing the same quality of synchronization as expected for horizontal synchronization.

Note that the terms horizontal and vertical in this context are not arbitrarily chosen, but correspond intuitively to the same terms used in the context of deltas. A vertical delta $G \xleftarrow{\delta^-} \Delta \xrightarrow{\delta^+} G'$ focuses on going from one triple to another, possibly on different levels of abstraction. A horizontal delta $\Delta S \xleftarrow{\delta^S} \Delta C \xrightarrow{\delta^T} \Delta T$, on the other hand, focuses more on delta propagation, extending, for example, a source delta to all other domains.

Finally, for presentation purposes, we have kept things simple and avoided complex topics that will be handled in detail in later chapters of this thesis. These include *application conditions* for TGG rules and *constraints* in both domains, which can formally be directly lifted to **Tri** but complicate the derivation process of TGG-based synchronizers. It must also have been painfully obvious to the reader that we have sidestepped all attribute manipulation in the TGG rules for the (simplified) running example. In practice, *especially* for vertical synchronization, attribute manipulation is crucial and will be handled with a novel and flexible TGG extension introduced with this thesis.

A FRAMEWORK FOR MODEL SYNCHRONIZATION

In this chapter, based on the ideas and concepts from [3, 5] by Anjorin et al., a framework for organizing and structuring model synchronization chains is presented. The framework is first of all discussed on a technology and standard agnostic level in Sect. 3.1, before it is specialized for TGGs and a complementary choice of auxiliary transformation languages and technologies in Sect. 3.2. Finally, in Sect. 3.3 the proposed approach is applied to the CME case study, which was introduced in Chap. 1.

By establishing this framework, this chapter provides CONTRIBUTION I addressing CHALLENGE I of this thesis as identified in Sect. 1.3:

Provide a framework for designing model synchronization chains, which combines a BX language with standard parser, unparser, and diff technologies in a systematic, standardized fashion, allowing for a (partial) reuse of generic adapters and existing solutions.

3.1 THE eMOFLON CODE ADAPTER (MOCA) FRAMEWORK

In the following, the term *platform* will be used to refer to the final step in a given vertical transformation chain. Depending on the application scenario, this might be a set of textual files (XML files, configuration files, property files, or code in a programming language), folder and file hierarchies (structures in a file system), an engineering tool with internal data structures that can be manipulated via an Application Programming Interface (API), or very simple and typically generic tree-like structures. In other words, in a vertical transformation, the platform is on the lowest level of abstraction, while the *target model* is on the highest level of abstraction.

Figure 3.1 depicts the eMoflon Code Adapter (MOCA) framework for organizing the components necessary for a bidirectional model-to-platform transformation. This framework does not prescribe any concrete technologies, transformation languages, or modelling standards. Note that for presentation purposes, the abstraction levels are arranged from left (low-level) to right (high-level).

The main idea of the framework is to support the integration of a BX language/tool with existing technologies for handling the platform. This is accomplished via a strict separation of the transformation into two distinct parts:

- (i) A platform-to-tree transformation, and (ii) a tree-to-model transformation.

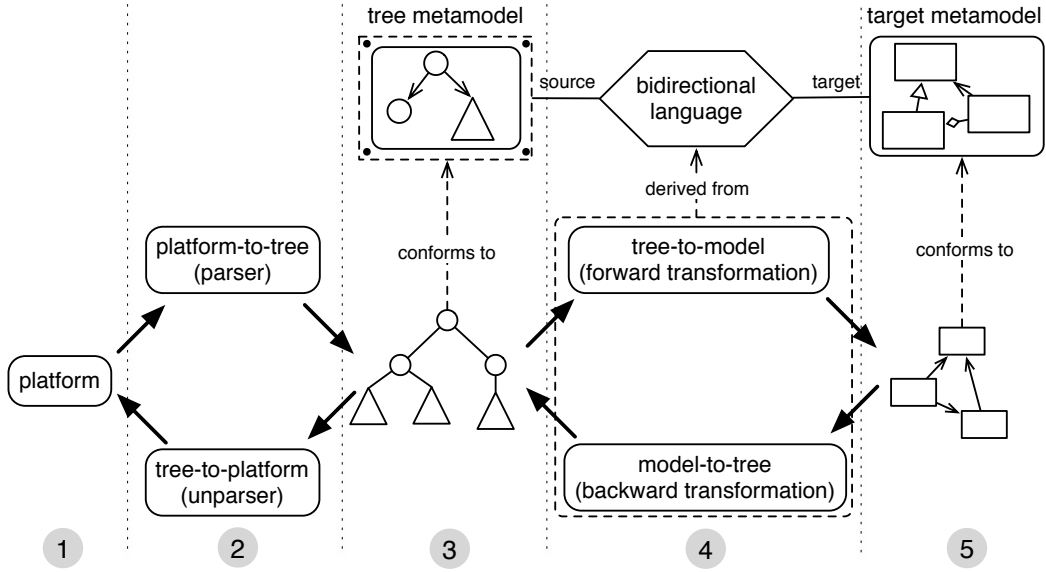


Figure 3.1: The eMoflon code adapter (MOCA) framework

The *platform* ① is transformed via a *parser* ② to a simple tree structure ③. This tree structure should be a minimal abstraction of the platform which is nonetheless accessible to the chosen bidirectional transformation language ④. Trees are transformed back to the platform via an *unparser* ② which typically linearizes the tree structure, and adds platform details that were abstracted away by the parser.

A crucial point is to keep the parser and unparser *as simple as possible*. This first step is often not worth supporting with a bidirectional language, and, if it is kept to a bare minimum, almost all complexity can be shifted to the model-to-tree transformation, which can be appropriately handled with a suitable bidirectional transformation language ④.

In an MDE context, the tree should be a very simple structure, which nonetheless already conforms to the modelling standard as required by the bidirectional transformation language and the target model ⑤. As almost all standard parsers are context-free,¹ “very simple” usually means acyclic and homogeneous with respect to typing (i.e., very few or even only a single “node” type is used).²

The set-up prescribed by this framework on a technology and standard agnostic level, already has a number of advantages:

SEPARATION OF CONCERNS:

A strict separation of platform *comprehension* and *generation* ② from the actual *transformation* ④ positively affects maintainability and productivity as the (un)parser can be kept very simple and be replaced without having to change the transformation.

¹ Mainly due to efficiency [81].

² Simplifies configuration of the parser and tree construction [82].

Furthermore, the bidirectional language can operate on a tree structure without irrelevant details of the textual representation leading to a simpler transformation with the clear tasks of: (i) adding appropriate typing information and (ii) deducing context-sensitive relations to obtain the target model ⑤.

A CLEAR INTERFACE TO DIFFERENT (UN)PARSER TECHNOLOGIES:

Establishing a simple tree structure for the bidirectional transformation consolidates XML and different abstract syntax trees produced by parsers. Even the directory and file structure can be embedded in the tree structure if it is relevant for the transformation. This positively affects the generality of the approach as support for new (textual) formats, e.g., JSON, can be added via generic adapters without having to change the choice of bidirectional language or other standards and technologies in the chain. Text separated into multiple files and folders can also be handled elegantly, even using multiple (un)parsers, as the resulting homogeneous sub-trees can be easily joined together to form the single input tree for the BX language.

Demanding only a semi-structured, i.e., hierarchical structure, greatly simplifies the task of parsing and unparsing, and clearly shifts most of the complexity to the tree-to-model transformation, which can be supported with a bidirectional language. This applies the right tool for the right job and also allows for using standard parser and unparser technology via simple adapters, which can be easily replaced. This has a positive affect on both maintainability and productivity.

MODULARITY:

In general, the modular structure of the framework enables a high level of reuse and exchangeability of the platform, parser and/or unparser, the model-to-tree transformation, the target metamodel and the modelling standard without affecting all other components. This positively affects maintainability as components are stable, productivity due to possible reuse, e.g., of existing (un)parsers, and generality, as at least parts of the system can be ported to a different platform or standard.

POTENTIAL FOR OPTIMIZATION:

As the tree metamodel ③ is fixed³ for a given modelling standard, generic support for this metamodel can be provided.

This support could include a suitable concrete syntax, editor support, or, more importantly, runtime optimizations such as caching and building up auxiliary index structures to substantially speed-up working with instances of the tree metamodel. Such optimizations are typically non-trivial to implement for a metamodel, and only become affordable if this metamodel does not constantly change and evolve.

³ Indicated in the diagram by depicting the tree metamodel in a frame that is “nailed” in place.

3.2 A TGG-BASED REALIZATION OF MOCA

Figure 3.2 depicts a realization of the MOCA framework with TGGs as the chosen bidirectional language.

With a focus on model-to-text transformations, the platform is now a directory structure containing multiple files in potentially different textual formats. This is denoted by the different shading (white, grey, black) used in ①.

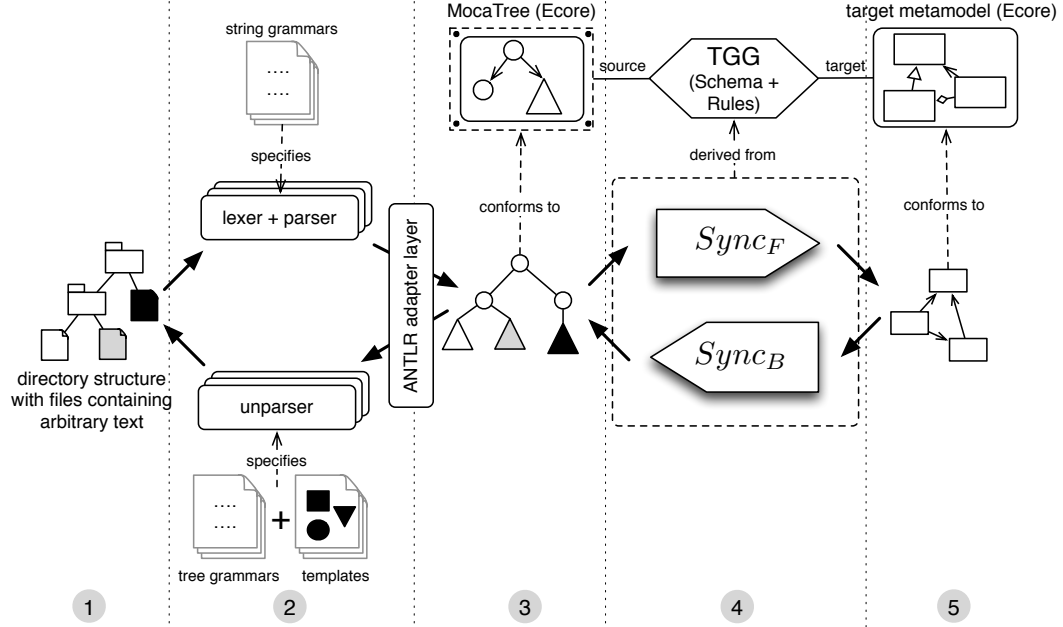


Figure 3.2: A TGG-based realization of MOCA

For each textual format, a pair of *lexer* and *parser* is used to transform the contents of a file to a tree. The parser generator ANTLR [81] is chosen as a mature, stable tool that can automatically derive a lexer and parser in Java from a high-level *string grammar*. From the input text, the lexer creates a stream of *tokens*, which is then used to create a tree by the parser.

To transform trees back to the textual format, the same tool ANTLR is used, this time generating an unparser from a *tree grammar* and a set of *templates*. A tree grammar is basically the “inverse” of a string grammar, with each rule parsing a tree fragment instead of creating it. The relatively simple template language StringTemplate [82] is used, which enforces a strict model-view separation, i.e., the templates are true views or *decorations* of the tree without any complex logic [80]. In this case, the templates are used to add static (default) text that was abstracted away by the parser. To support the direct usage of ANTLR, an *adapter layer* is provided to convert ANTLR trees to Eclipse Modeling Framework (EMF)-conform models and vice-versa. For each parser/unparser technology to be used, e.g., when dealing with XML files or folder/file structures, a corresponding adapter layer must be provided.

As an interface to the EMF-based modelling space, the tree metamodel (Moca-Tree) depicted in Fig. 3.3 is used. A MocaTree (Fig. 3.2::③) is an acyclic, labelled graph consisting of Nodes each with a name and an index. The latter indicates the position of the node in the list of children of its parent node. Note that a few further concepts are present in the metamodel such as Files, Folders, Attributes, and Text nodes that cannot have any children. These elements are necessary to handle XML files and structures in the file system. MocaTree is clearly chosen to be as simple as possible, i.e., a minimal abstraction. Further concepts common in programming languages such as scoping and references are intentionally *not* explicitly modelled as such analyses are meant to be implemented on the target model using MDE technology. The observant reader might have noticed that indices are used to sort children nodes in MocaTrees and not, e.g., next edges. The practical reason for this design decision is that it is currently easier to ignore irrelevant attribute values with TGGs than to ignore structural links in models, as the latter are also used to control the transformation process (cf. Def. 54). As next edges are, however, advantageous in many cases, either optionally enabling/disabling next edge creation as required, or extending TGGs to handle such “derived” links in general, would be useful and should be investigated as future work.

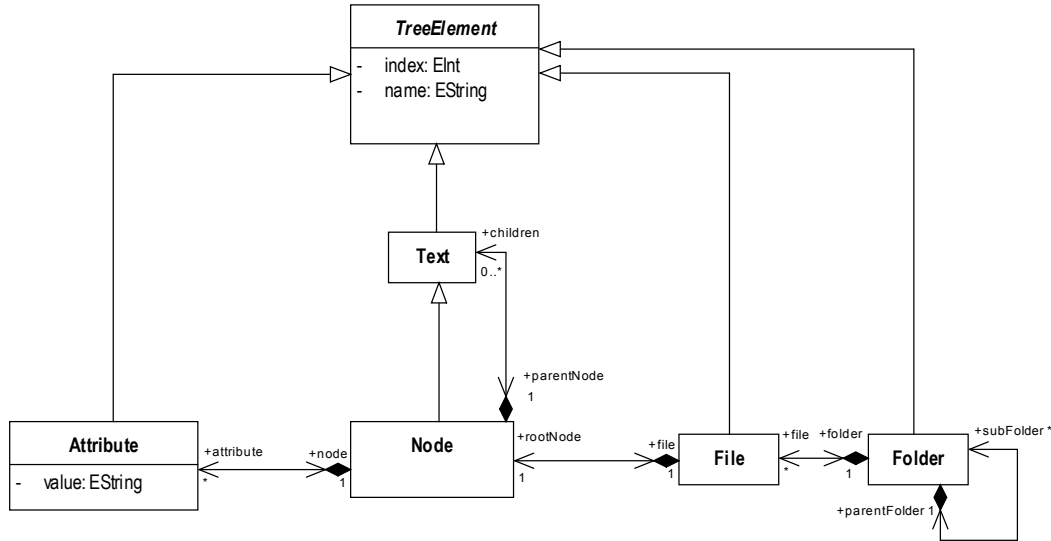


Figure 3.3: Tree metamodel used as an interface to the EMF modelling space

With the platform now lifted to the required modelling standard, TGG rules (Fig. 3.2::④) can be used to specify synchronizers that operate on trees and instances of the target metamodel.

Example 12 (*A code adapter for the running example*).

The text-to-tree transformation for the running example is depicted in Fig. 3.4 for a simple CLS file. As a first task, the transformation abstracts from static text such as commas (”,”) and keywords, i.e., `GOTO` in the CLS file could be renamed to `MOVE` and the tree would remain the same. Note that some elements are completely ignored during tree construction such as `END-OF-PATH` and all

whitespace. The transformation also decides how detailed the text comprehension must be. Comments, for example, are simply copied line by line into the tree without any further analysis. Whole sections of text can be handled in this manner as a means of abstraction.

The second task of the transformation is to organize the extracted information into a hierarchical structure. When using TGGs, it is usually helpful to *normalize* the tree structure. An example of this are the `COMMENT_NODE`s in Fig. 3.4. In the tree, every command has a `COMMENT_NODE`, even if there was actually no corresponding comment in the textual file. Similarly, one could also have introduced an `ARGS_NODE` to group the arguments of each command. This is not absolutely necessary; there are always a fixed number of arguments for each command in the CLS file, and the position of the node in the list of children of its parent node can be used to uniquely identify the argument.

Finally, note that the empty line in the input file is removed when unparsing the tree. This is an example of information loss already in the text-to-tree transformation. If this is to be absolutely avoided, the parser can also be made to be whitespace sensitive, i.e., such empty lines can be preserved in the tree (but most probably not in the target model!).

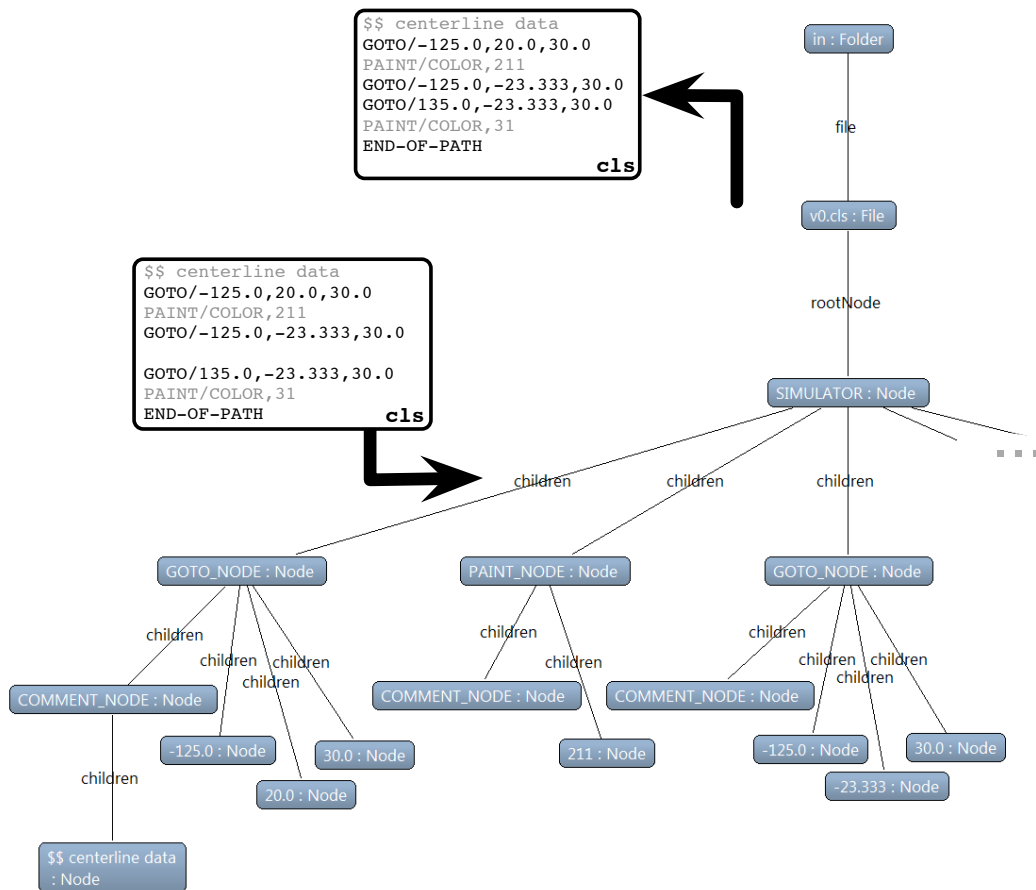


Figure 3.4: Text-to-tree roundtrip for the running example

The lexer and parser used to accomplish this text-to-tree transformation are depicted in Fig. 3.5. Both lexer and parser grammars are string grammars in context-free Extended Backus-Naur Form (EBNF). ANTLR allows mixing in arbitrary Java code enclosed in curly brackets after a rule. For example, after the rule WS for recognizing whitespace, the `skip()` method of the lexer ① is invoked to make sure that the whitespace token is thrown out of the generated stream of tokens. User-defined methods can be invoked in the same manner.

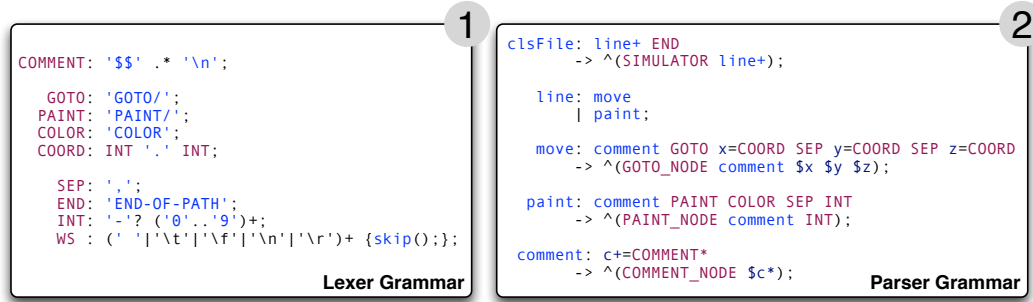


Figure 3.5: Lexer and parser string grammars for the running example

In addition to rules for recognizing textual fragments, the parser ② contains directives after the arrow symbol “->” for tree generation. The directive \wedge (SIMULATOR line+), for example, creates a tree with SIMULATOR as its root node and the recursively created line sub-trees as children (cf. Fig. 3.4).

The tree grammar and template rules necessary for unparsing the tree are depicted in Fig. 3.6. A tree grammar ③ consists of rules, this time with tree fragments as the structure to be recognized, and template evaluation as the directives to be executed when a rule matches. For example, after recognizing the root node SIMULATOR and its line children, the template `clsFile` is evaluated, passing the list of lines as a parameter.

Finally, a string template such as ④ also consists of rules describing how the parameters of each template rule are to be transformed to text. For example, the template rule `clsFile` renders each line separated by “\n”, and finalizes the file with the static text fragment END-OF-PATH.

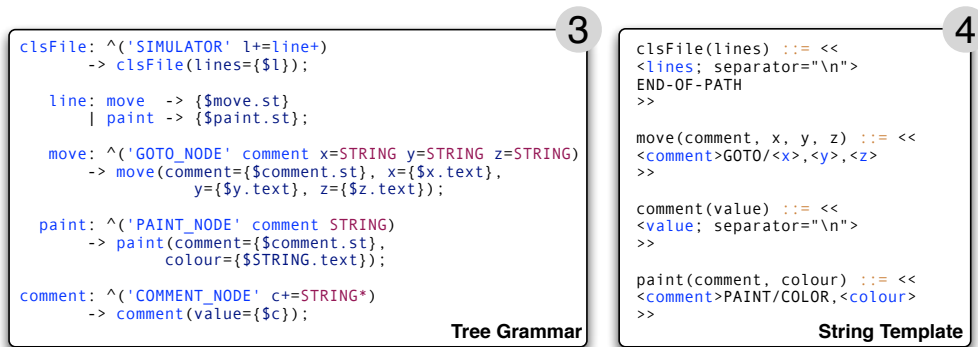


Figure 3.6: Tree grammar and string template for the running example

As this thesis is not about ANTLR, string grammars, templates, or tree grammars, further details are not necessary and the interested reader is referred to [81, 82] for a complete documentation of ANTLR and all supported languages. This example is just to show how a well-known parser generator can be integrated into the MOCA framework. Other parser generators, as well as XML reader/writers, for instance, can be integrated analogously.

Figure 3.7 depicts the final step in the chain, the tree-to-model transformation that is used to synchronize trees with their corresponding simulator models. Before we can specify this transformation with TGGs, however, we still need to cover attribute manipulation in TGG rules, which will be the topic of Chap. 4.

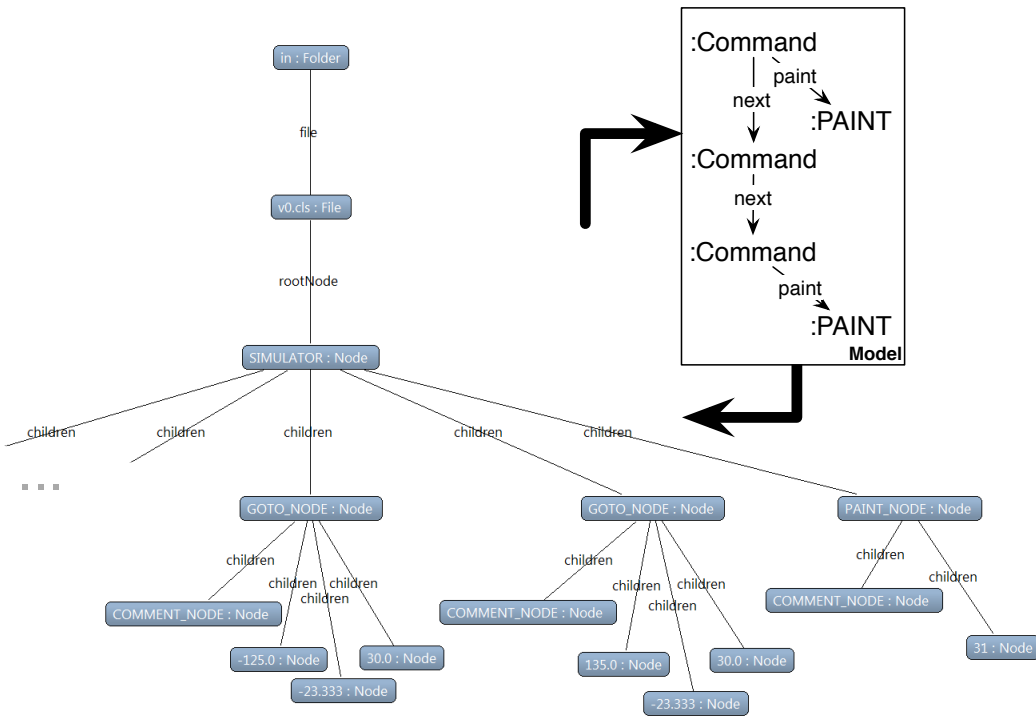


Figure 3.7: Required tree-to-model transformation for the running example

Advantages of this TGG-based realization of the MOCA framework include:

HOMOGENEITY VIA COMPLEMENTARY LANGUAGES:

All chosen languages, i.e., string grammars, tree grammars combined with string templates, and TGGs, are *rule-based* and *declarative*. With such a mix of transformation languages, a high level of homogeneity is attained by supporting a common rule-based thinking in *patterns*. This positively affects maintainability and productivity as there is no disturbing shift in paradigm. Trained skills in one language can also be transferred to all others.

FORMAL PROPERTIES AND GUARANTEES:

By separating the transformation into different steps, the formal properties of the different individual languages still hold for the corresponding step. For example, the worst-case runtime complexity for LL* parsing (worstcase $O(n^2)$, in practice actually much less [81]) holds for tree generation, while TGGs guarantee polynomial runtime in model size [67] for the tree-to-model transformation. Depending on the application scenario, such runtime efficiency can be crucial for scalability. Furthermore, TGGs also guarantee that the derived transformations are *correct* with respect to the specified TGG, i.e., only a single specification is used, which positively affects maintainability.

FLEXIBLE FALLBACK TO JAVA:

Practical problems can almost never be *completely* solved with a DSL [37, 97]. For example, even if a large part of a transformation can be specified with TGGs, certain parts, especially low-level attribute manipulation, must be specified directly in Java. All languages used in the presented MOCA framework support a *fallback* to a more expressive language when necessary. ANTLR, for instance, offers *syntactic* and *semantic predicates* in string and tree grammars [81], which can be used to embed Java statements to support the lexer/parser.

ANTLR and eMoflon are both completely generative, i.e., they map specifications to Java code which also simplifies mixing in hand-written code. This positively affects generality, as basically any problem can be tackled that could also have been solved directly in a general purpose language, in this case Java.

ITERATIVE WORKFLOW:

Finally, an iterative workflow is possible as the target metamodel can be iteratively refined. In each step, more parts of the tree can be handled by new TGG rules until the transformation is complete. The platform can also be handled in an iterative manner, e.g., by using regular expressions to “filter” the textual files in the first iterations instead of a parser. ANTLR also supports this with a “fuzzy” parsing mode that ignores all content that cannot be parsed, i.e., parses these parts as a string block without further processing/structuring. An iterative workflow improves productivity as most mistakes can be found early enough in the development process.

3.3 APPLYING MOCA TO CME

In this section, the complete process involved in applying the MOCA framework to a model synchronization application scenario is discussed. Two strategies, an *elaborate* and a *lightweight* strategy, are presented and compared based on practical experience gained from applying MOCA to the CME case study presented in Chap. 1.

3.3.1 An Elaborate Approach to Applying MOCA

Figure 3.8 depicts the first *elaborate* approach, consisting of two vertical, and one horizontal TGG. The first step is to establish suitable abstractions for CLS and MPF files, represented as metamodels ① and ②, respectively. These metamodels define the parts of both formats that are to be considered relevant for the synchronization. Especially in an iterative approach, these abstractions typically start off by being minimal, but grow with time to cover a certain subset of the respective language.

The next step is to provide (un)parsers to be able to transform CLS ③ and MPF files ④ to instances of the common tree metamodel (in this case MocaTree).

These trees can now be transformed to instances of the respective metamodels using synchronizers derived from the vertical TGGs ⑤ and ⑥. The induced consistency relation is as follows: If a synchronizer derived from the horizontal TGG ⑦ can be used to extend a pair of source and target models to a consistent triple by creating a correspondence model ⑧, then the pair of source and target models is consistent.

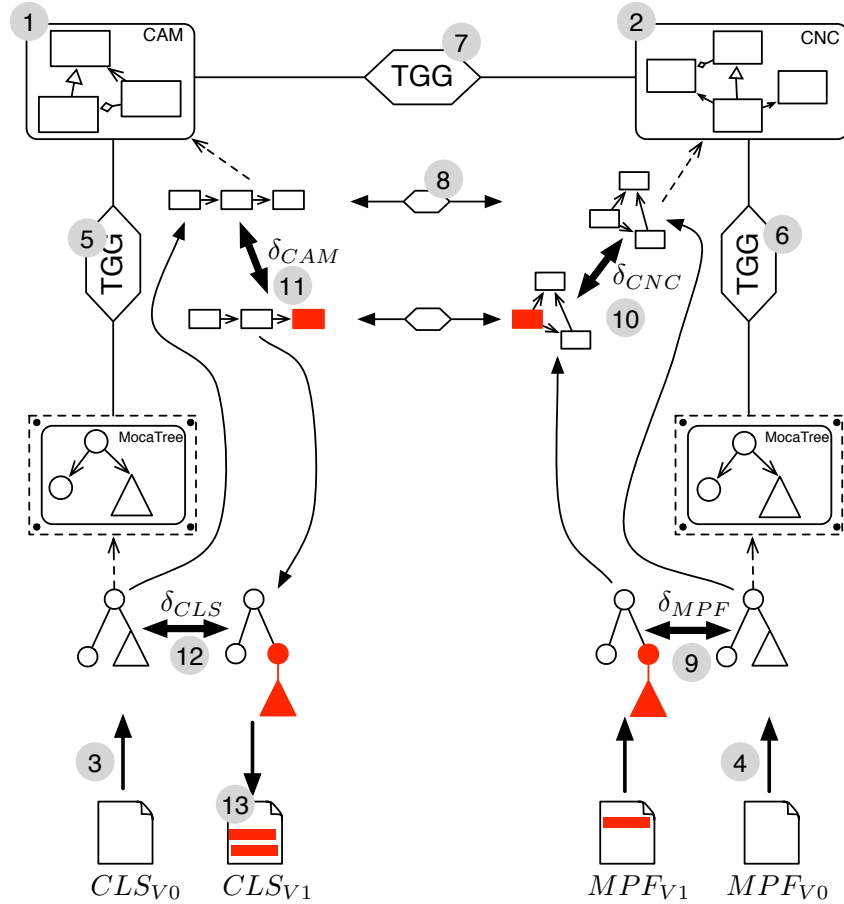


Figure 3.8: Applying MOCA to the CME case study

As soon as an initial consistent triple has been established, the workflow for propagating MPF deltas⁴ is now as follows (analogously for CLS deltas):

- (i) The MPF file MPF_{V0} is edited resulting in a new file MPF_{V1} .
- (ii) If this is done “offline” without any special editors or a listener framework, the explicit delta representing the change has to be determined by comparing both versions. Although this can be done on a textual level via a *text diff*, it is probably better to use a *tree diff* to construct δ_{MPF} ⑨ by comparing the old tree from MPF_{V0} , to the new tree from MPF_{V1} .
- (iii) The delta δ_{MPF} is propagated to a delta δ_{CNC} ⑩ using the synchronizer derived from ⑥.
- (iv) The delta δ_{CNC} is propagated to a delta δ_{CAM} ⑪ using the synchronizer derived from ⑦.
- (v) The delta δ_{CAM} is propagated to a delta δ_{CLS} ⑫ using the synchronizer derived from ⑤.
- (vi) Finally, the updated CLS tree is unparsed to a new updated version of the CLS file CLS_{V1} ⑬, which is now consistent with MPF_{V1} .

3.3.2 A Lightweight Approach to Applying MOCA

An alternative *lightweight* approach can be taken, if the assumption holds that both textual formats can be transformed (with acceptable effort) to the exact same abstraction. If this is the case, then the chain of three TGGs required in Fig. 3.8 can be reduced to just two as depicted in Fig. 3.9. The single metamodel ①⁵ represents the common abstraction that is to be extracted from both CLS and MPF files. As in the prior approach, the process starts by parsing the CLS file CLS_{V0} ② and the MPF file MPF_{V0} ③ to their respective trees. These trees are then lifted to instances of the common abstraction using ④ and ⑤.

The induced consistency relation between CLS and MPF files is now as follows: if a pair of CLS and MPF files leads to isomorphic instances ⑥ of the common target metamodel, then they are consistent. In other words, with respect to the common abstraction ①, the models extracted from the files are identical (isomorphic).

The workflow for propagating MPF deltas is as follows (analogously for CLS):

- (i) The MPF file MPF_{V0} is edited resulting in a new file MPF_{V1} .
- (ii) As in the previous workflow (Fig. 3.8), if this is done “offline”, a *tree diff* is required to construct δ_{MPF} ⑦ by comparing the old tree from MPF_{V0} , to the new tree from MPF_{V1} .

⁴ Delta in the sense of Def. 17.

⁵ For the CME project, the CAM metamodel was chosen to be the common abstraction. In general, however, this could also have been the CNC metamodel or a totally different metamodel “in-between” the abstractions of CAM and CNC files.

- (iv) The delta δ_{CAM} is propagated to a delta δ_{CLS} ⑨ using the synchronizer derived from ④.

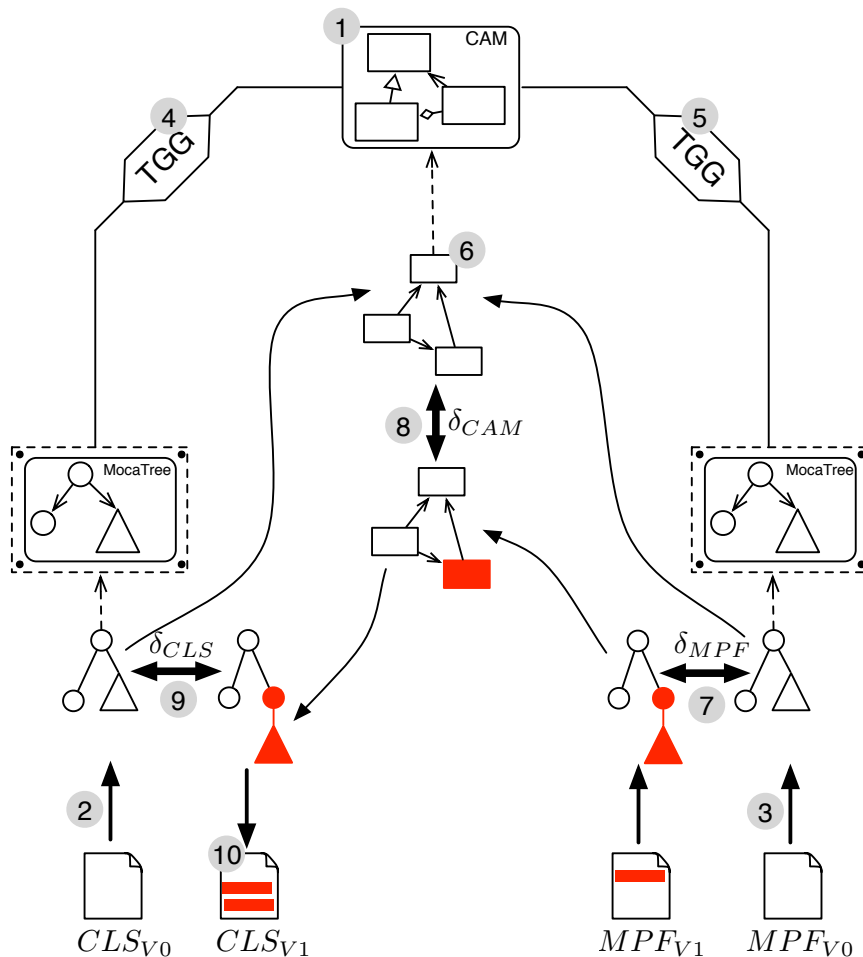


Figure 3.9: Alternative MOCA-based approach to the CME case study

If both vertical TGGs ④ and ⑤ only incur information loss in the tree-to-model direction, meaning that the abstraction is a true view of the respective trees, the process can be simplified further by ignoring (i.e., not calculating) δ_{MPF} , and recreating the new target model from scratch. A *model diff* is then used to construct δ_{CAM} from two versions of the target model. The final step *must* be incremental, however, as the CLS tree cannot be recreated completely from the updated target model. The main advantage of this further simplification is that the diff is only required on the high-level of abstraction. This typically works better in practice as there is less irrelevant information that would otherwise obscure comparison results.

3.3.3 A Comparison of Both Approaches to Applying MOCA

To conclude this section, the following discussion provides an overview of salient points to consider when applying MOCA to an application scenario:

- Constructing a satisfactory delta from two versions of a model or tree is challenging in practice. If possible, deltas should be “recorded online” as changes are made, e.g., using a suitable editor or a listener framework. Delta reconstruction can be error-prone, non-incremental, relatively inefficient, and can swiftly become the bottle-neck in a MOCA-based model synchronization chain [73].
- It is important to note that the elaborate approach (Fig. 3.8) and the lightweight alternative (Fig. 3.9) differ in their induced concept of consistency. With the former, a materialized witness for consistency is available in form of the correspondence model that is created for consistent source and target models. This is not the case with the latter, as two instances of the target model are produced that are equivalent with respect to a model diff. It is, therefore, more difficult to “explain” or “see” why exactly the two files are consistent as this is hidden in the transformations. In practice, the correspondence model provides a means for users to comprehend the results of a synchronization. If this is paramount, then the first approach should be preferred.
- The lightweight approach (Fig. 3.9) reduces the synchronization steps required from three to two. An even more important point here is that the elaborate approach requires a special case of model integration. When starting the process, initial versions of both files are present, and a first version of the correspondence model (Fig. 3.8:⑧) must be constructed given both models. This can be viewed as a special case of integrating both a source *and* target delta. As this is out-of-scope for this thesis, the complete CME case study was implemented with the lightweight approach. Recall that the workflow of the lightweight approach avoids this by enforcing a single metamodel and performing a model diff instead of model integration.
- In case of the lightweight approach, the “semantic distance” of both textual formats from the abstraction is obviously greater than for the elaborate approach. This can result in an increased complexity of the required TGGs and also potentially violates the principle of separation of concerns.
- Finally, the vertical TGGs established for each textual format in the elaborate approach can be possibly reused in a different context, by swapping the horizontal TGG connecting both metamodels. In this manner, a generic “code adapter” can be established for each format, which is independent of a particular synchronization scenario (at least to a certain extent).

Both approaches to applying MOCA have advantages and disadvantages. Depending on the application scenario, it is also possible to start with one approach and then either split the metamodel into two different abstractions, or merge the source and target metamodels to form a single abstraction.

SPECIFICATION OF TGG-BASED SYNCHRONIZERS

In this chapter, the language extensions to TGGs, identified and implemented in the context of this thesis, will be presented and discussed in detail.

To increase expressiveness without destroying the simplicity and elegance of TGG rules, the first two extensions allow a controlled fallback to a GPL.

Attribute conditions, which can be implemented in any language, are formalized as black-box predicates over attribute values and are used as additional application conditions for TGG rules. This extension, together with a formalization of attribution for typed graphs is presented in Sect. 4.1 based on [7] by Anjorin et al.

Dynamic conditions are formalized as black-box functions evaluated during rule application and are also used as additional application conditions for TGG rules. Analogously to normal conditions, dynamic conditions attempt to extend a match to a larger graph that is, however, determined dynamically (at rule application and not rule specification time) by a black-box function that can be implemented in any language. This increases the expressiveness of TGG rules, and also provides a means of optimizing crucial parts of a transformation by replacing relevant sub-patterns by an efficient dynamic condition leveraging, e.g., caching mechanisms. Initially demonstrated in [2] by Anjorin et al., this is presented in Sect 4.2.

In practice, not only expressiveness but also *maintainability* is crucial when tackling real-world applications with TGGs. To improve the modularity of TGG rules, therefore, the flexible concept of *rule refinement* based on [10] by Anjorin et al. is presented in Sect. 4.3.

With these new TGG language features to increase expressiveness, and a modularity concept to improve maintainability, this chapter provides CONTRIBUTION II addressing CHALLENGE II of this thesis as identified in Sect. 1.3:

Provide a transformation language that is at the same time high-level enough to support productivity and maintainability, and is also expressive, efficient and scalable enough for real-world applications.

4.1 FLEXIBLE ATTRIBUTE MANIPULATION IN TGG RULES

Although graphs are extended in the following to incorporate attribute values, *attribute conditions*, i.e., demanding that a condition holds for certain attribute values, are only introduced as a form of application conditions for attributed triple rules. This means that attributed graphs, and consequently graph conditions (graph constraints and application conditions) cannot have attribute conditions. This is

a limitation that simplifies the formalization and implementation but reduces the expressive power of construction techniques and static analyses (cf. Chap. 6).

4.1.1 Extending our Graph Concept to Cover Attribution of Graph Nodes

To introduce attributes to our current concepts of typed graphs and typed graph morphisms, the first step is to establish a suitable category from which attribute values will be taken. As attribute values are to be *typed*, e.g., of type `String` or `Integer`, the following definition introduces the concept of an *algebraic signature* according to [28].

To avoid confusion with the type morphism already introduced for typed graphs, the term *sort* instead of *type* is used for attributes. As the proposed language feature for complex attribute manipulation in TGG rules only requires predicates¹, i.e., operations of sort `Bool`, all definitions are simplified accordingly.

Definition 21 (*Predicate Algebraic Signature Σ*).

A *predicate algebraic signature* $\Sigma = (S, P)$ consists of a set S of sorts with a distinguished sort `Bool` $\in S$, and a set P of predicate symbols $p : (s_1, s_2, \dots, s_n) \rightarrow \text{Bool}$, where $s_i \in S$, $\forall i \in \{1, \dots, n\}$.

For brevity, *predicate algebraic signatures* will be referred to as *signatures*.

Example 13 (*A signature for our running example*). —————

To explain the introduced concepts in this chapter with a running example, we shall specify a simplified version of the vertical TGG required for the CLS-tree to model transformation illustrated in Fig. 3.7.

The following signature is sufficient to type all attributes and predicates required for the transformation:

$$\begin{aligned}\Sigma &= (S, P) \\ S &= \{\text{Bool}, \text{String}, \text{Integer}, \text{Float}\} \\ P &= \{\text{addSuffix}, \text{add}, \text{stringToNumber}\}\end{aligned}$$

The number and sorts of input and output arguments for all predicate symbols must also be specified:

$$\begin{aligned}\text{addSuffix} : & \quad (\text{String}, \text{String}, \text{String}) \rightarrow \text{Bool} \\ \text{add} : & \quad (\text{Integer}, \text{Integer}, \text{Integer}) \rightarrow \text{Bool} \\ \text{stringToNumber} : & \quad (\text{String}, \text{Float}) \rightarrow \text{Bool}\end{aligned}$$

¹ Relations between attributes are stated declaratively as a set of direction independent “facts” (predicates), which are then operationalized for different scenarios, i.e., forward, backward.

The same signature Σ can be implemented differently as long as the implementation conforms to the signature. This will be important when attributing models as opposed to metamodels or rules. Such implementations or realizations of a signature are called *algebras* and are formalized as follows.

Definition 22 (*Predicate Σ -Algebra A_Σ*).

A predicate Σ -Algebra $A_\Sigma = (S_A, P_A)$ over a signature $\Sigma = (S, P)$ is defined by:

- a set $S_A = \{A_s \mid s \in S\}$ of carrier sets A_s for each sort $s \in S$.
- a set $P_A = \{p_A \mid p \in P\}$ of predicates $p_A : A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n} \rightarrow A_{\text{Bool}}$ for each predicate symbol $p : (s_1, s_2, \dots, s_n) \rightarrow \text{Bool} \in P$.

For brevity, *predicate Σ -algebras* will be referred to as *algebras*.

Example 14 (*An algebra for our running example*).

An algebra is to be regarded as an interface to a GPL that supplies the carrier sets and realizations of all predicates. For our running example, the following algebra $A = (S_A, P_A)$, implementing the signature $\Sigma = (S, P)$ from Ex. 13, will be used to provide attributes for CLS trees.

$$\begin{aligned} A_{\text{Bool}} &= \{\text{true}, \text{false}\}, \quad A_{\text{Integer}} = \mathbb{Z}, \quad A_{\text{Float}} = \mathbb{R} \\ A_{\text{String}} &= \{\text{"SIMULATOR"}, \text{"GOTO_NODE"}, \text{"COMMENT_NODE"}, \dots\} \end{aligned}$$

The predicates are defined on the carrier sets according to Σ :

$$\begin{aligned} \text{addSuffix}_A &: A_{\text{String}} \times A_{\text{String}} \times A_{\text{String}} && \rightarrow A_{\text{Bool}} \\ \text{add}_A &: A_{\text{Integer}} \times A_{\text{Integer}} \times A_{\text{Integer}} && \rightarrow A_{\text{Bool}} \\ \text{stringToNumber}_A &: A_{\text{String}} \times A_{\text{Float}} && \rightarrow A_{\text{Bool}} \end{aligned}$$

with the following implementation given in pseudo Java code:

$$\begin{aligned} \text{addSuffix}_A(a, b, c) &\equiv c.\text{equals}(a + b) \\ \text{add}_A(i, j, k) &\equiv k == i + j \\ \text{stringToNumber}_A(a, i) &\equiv i == \text{Float.parseFloat}(a) \end{aligned}$$

Definition 23 (*Predicate Σ -Algebra Homomorphism*).

Given algebras A and A' over a common signature $\Sigma = (S, P)$, a predicate algebra homomorphism $f : A \rightarrow A'$ is a set $f = \{f_s \mid s \in S\}$ of mappings $f_s : A_s \rightarrow A'_s$ for each $s \in S$, such that $\forall p : (s_1, s_2, \dots, s_n) \rightarrow \text{Bool} \in P$, Fig. 4.1 commutes, i.e., $p_{A'} \circ (f_1, \dots, f_n) = f_{\text{Bool}} \circ p_A$.

For brevity, *predicate Σ -algebra homomorphisms* will be referred to as *homomorphisms*.

$\mathbf{Alg}(\Sigma) = (\mathbf{Ob}_{\mathbf{Alg}(\Sigma)}, \mathbf{Arr}_{\mathbf{Alg}(\Sigma)}, \circ_{\mathbf{Alg}(\Sigma)}, \mathbf{id}_{\mathbf{Alg}(\Sigma)})$ consists of:

- algebras $\mathbf{Ob}_{\mathbf{Alg}(\Sigma)}$,
- homomorphisms $\mathbf{Arr}_{\mathbf{Alg}(\Sigma)}$,
- for $A, A', A'' \in \mathbf{Ob}_{\mathbf{Alg}(\Sigma)}$, $f : A \rightarrow A'$, $g : A' \rightarrow A''$, $g \circ_{\mathbf{Alg}(\Sigma)} f : A \rightarrow A''$ is defined as a set $\{(g \circ_{\mathbf{Alg}(\Sigma)} f)_s \mid s \in S\}$ of maps $(g \circ_{\mathbf{Alg}(\Sigma)} f)_s : A_s \rightarrow A''_s$, where $(g \circ_{\mathbf{Alg}(\Sigma)} f)_s(x) := g(f(x))$ for $x \in A_s$,
- for $A \in \mathbf{Ob}_{\mathbf{Alg}(\Sigma)}$, $\mathbf{id}_A : A \rightarrow A$ is defined as a set of maps $\{(\mathbf{id}_A)_s \mid s \in S\}$, where $(\mathbf{id}_A)_s(x) := x$ for $x \in A_s$.

$$\begin{array}{ccccccc}
 A_{S_1} \times A_{S_2} \times \cdots \times A_{S_n} & \xrightarrow{p_A} & A_{Bool} \\
 \downarrow f_1 & & \downarrow f_{Bool} \\
 A'_{S_1} \times A'_{S_2} \times \cdots \times A'_{S_n} & \xrightarrow{p_{A'}} & A'_{Bool}
 \end{array}
 \quad \text{with} \quad \begin{array}{c} f_2 \quad \cdots \quad f_n \\ = \end{array}$$

Figure 4.1: Homomorphisms preserve the structure in algebras

After introducing homomorphisms as structure preserving maps between algebras, we now finalize the introduction of attributes by defining the corresponding category $\mathbf{Alg}(\Sigma)$.

Fact 7 ($\mathbf{Alg}(\Sigma)$ is a category).

$\mathbf{Alg}(\Sigma)$ with algebras as objects and homomorphisms as arrows forms a category.

Proof. (Sketch) For homomorphisms g, f , $g \circ_{\mathbf{Alg}(\Sigma)} f$ can be shown to be again a homomorphism by arguing with commuting squares. Associativity and identity follow directly from Def. 23. The reader is referred to, e.g., Sect. 3.2.7 in [86]. \square

Algebras are used to provide the attribute values in a graph. The concept of a graph is extended by a set of *attribute nodes* (infinite in general) taken from an algebra, and *attribute edges* used to connect the normal nodes in the graph to their attribute nodes. This leads to the notion of *attributed graphs*.

Definition 24 (*Attributed Graphs and Attributed Graph Morphisms*).

Given a signature $\Sigma = (S, P)$, an *attributed graph* $AG = (G, V_A, E_A, s_A, t_A)$ over an algebra A , consists of a graph $G = (V, E, s, t)$, a set V_A of attribute nodes, a set E_A of attribute edges, and two functions $s_A : E_A \rightarrow V, t_A : E_A \rightarrow V_A$ that assign each attribute edge $e_A \in E_A$ a source node $s_A(e_A) \in V$ in the graph, and a target attribute node $t_A(e_A) \in V_A$. The set V_A of attribute nodes is formed by taking the disjoint union of all carrier sets A_s of the algebra A , i.e., $V_A := \uplus_{s \in S} A_s$.

Given attributed graphs $AG = (G, V_A, E_A, s_A, t_A)$ over A , and $AG' = (G', V_{A'}, E_{A'}, s_{A'}, t_{A'})$ over A' , an *attributed graph morphism* $f : AG \rightarrow AG'$ is a tuple $f = (f_G, f_{V_A}, f_{E_A})$ of a graph morphism $f_G : G \rightarrow G'$, and functions $f_{V_A} : V_A \rightarrow V_{A'}$, $f_{E_A} : E_A \rightarrow E_{A'}$, such that Fig. 4.2::a commutes, and f_{V_A} is a homomorphism, i.e., the restrictions $(f_A)_s := f_{V_A}|_{A_s}$ of f_{V_A} to A_s form a homomorphism $f_A = (f_A)_s : A \rightarrow A'$.

AGraphs = $(\text{Ob}_{\mathbf{AGraphs}}, \text{Arr}_{\mathbf{AGraphs}}, \circ_{\mathbf{AGraphs}}, \text{id}_{\mathbf{AGraphs}})$ consists of:

- attributed graphs $\text{Ob}_{\mathbf{AGraphs}}$,
- attributed graph morphisms $\text{Arr}_{\mathbf{AGraphs}}$,
- for $AG, AG', AG'' \in \text{Ob}_{\mathbf{AGraphs}}$, $f : AG \rightarrow AG'$, $g : AG' \rightarrow AG''$, $g \circ_{\mathbf{AGraphs}} f : AG \rightarrow AG''$ is defined as:
 $g \circ_{\mathbf{AGraphs}} f := (g_G \circ_{\mathbf{Graphs}} f_G, g_{V_A} \circ_{\mathbf{Alg}(\Sigma)} f_{V_A}, g_{E_A} \circ_{\mathbf{Sets}} f_{E_A})$.
- for $AG \in \text{Ob}_{\mathbf{AGraphs}}$, $\text{id}_{AG} : AG \rightarrow AG$ is defined as $(\text{id}_G, \text{id}_{V_A}, \text{id}_{E_A})$, where $\text{id}_G \in \text{id}_{\mathbf{Graphs}}$, $\text{id}_{V_A} \in \text{id}_{\mathbf{Alg}(\Sigma)}$, $\text{id}_{E_A} \in \text{id}_{\mathbf{Sets}}$.

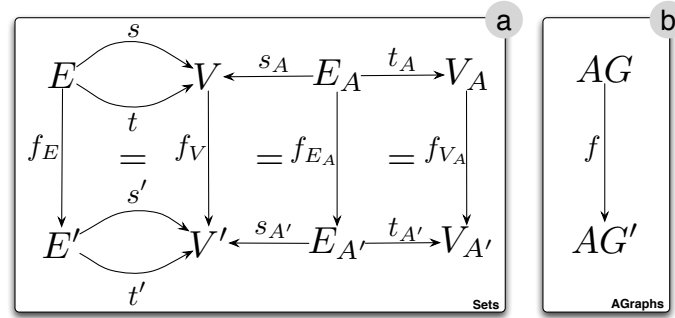


Figure 4.2: Attributed graph morphisms are structure preserving maps

To simplify the construction of pushouts in **AGraphs**, we consider a special class $\overline{\mathcal{M}}$ of monomorphisms that are additionally identities on the underlying algebra. Although this is a strong restriction, it turns out to be sufficient to formalize rule-based transformation in **AGraphs**.

Note that only rule morphisms have to be in $\overline{\mathcal{M}}$ as pushouts are required for rule application. All other attributed graph morphisms (type, match morphisms) do not have to be in $\overline{\mathcal{M}}$.

Definition 25 (Class $\overline{\mathcal{M}}$ in **AGraphs**).

In **AGraphs**, the class $\overline{\mathcal{M}}$ of attributed morphisms is defined as the class of all monomorphisms $f = (f_V, f_E, f_{V_A}, f_{E_A})$ with $f_{V_A} = \text{id}_{V_A}$. This means that the homomorphism corresponding to f_{V_A} is the identity on the underlying algebra, i.e., $\overline{\mathcal{M}}$ -morphisms between attributed graphs preserve the algebra.

The following fact shows that **AGraphs** with $\overline{\mathcal{M}}$ as defined above, can now be used as the category in Def. 8 and 9, i.e., rule-based model transformation can be lifted to *attributed* graphs. The interested reader is referred to Fact 8.12 in [28] for a detailed proof in a more general setting.

Fact 8 (**AGraphs** is a category with pushouts along $\overline{\mathcal{M}}$ -morphisms).

AGraphs with attributed graphs as objects and attributed graph morphisms as arrows forms a category.

Given attributed morphisms $r : L \rightarrow R \in \overline{\mathcal{M}}$, and $m : L \rightarrow G$ in **AGraphs**, a pushout $(r' : G \rightarrow G', m' : R \rightarrow G')$ can be constructed in **AGraphs** with $r' \in \overline{\mathcal{M}}$.

Proof. (Sketch) Soundness of composition can again be argued with commuting squares of the form depicted in Fig. 4.2, and soundness of composition already shown for **Graphs**, **Sets**, and **Alg**(Σ). Similarly, associativity and identity are inherited componentwise from **Graphs**, **Sets**, and **Alg**(Σ).

Given $r = (r_G, r_{V_A}, r_{E_A}) : L \rightarrow R$ and $m = (m_G, m_{V_A}, m_{E_A}) : L \rightarrow G$, the pushout construction in **AGraphs** is defined componentwise: (r'_G, m'_G) in **Graphs**, (r'_{E_A}, m'_{E_A}) in **Sets**, and (id, m_{V_A}) in **Alg**(Σ). As the construction is componentwise, Fig. 4.3 is commutative and the universal pushout property follows from the pushout construction in each component. By construction, r' is obviously in $\overline{\mathcal{M}}$ as its data mapping is the identity. The interested reader is referred to Fact 8.12 in [28] for a detailed proof. \square

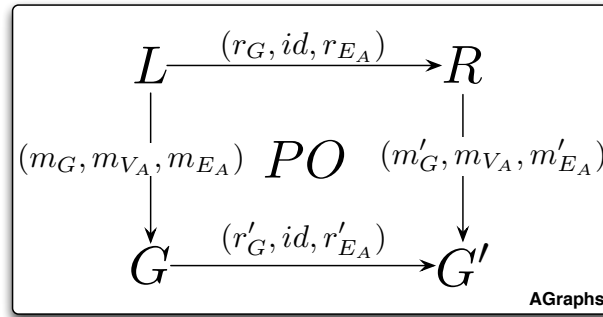


Figure 4.3: Pushout construction along $\overline{\mathcal{M}}$ -morphisms in **AGraphs**

Example 15 (*Tree model as an attributed graph*). —

Figure 4.4 depicts an attributed graph $AG = (G, V_A, E_A)$ in a detailed notation to the left, and in a compact notation for attributed graphs to the right. For presentation purposes, the latter will be used whenever possible.

In the formal notation, note that the source and target functions between E_A/G , and E_A/V_A , respectively, are depicted by denoting the attribute edges in E_A as arrows with a dashed line going from graph nodes in G to data nodes

in V_A . In the compact notation for attributed graphs, this is denoted as an entry *inside* the graph node, e.g., name: "SIMULATOR".

A further point to note here is that attribute nodes can (unfortunately) have *multiple* values according to the formalization. E.g., there is nothing preventing the graph node `rootNode` from being connected via multiple name attribute edges to different attribute nodes such as "SIMULATOR" and "ROOT". This is often not wanted but must be explicitly forbidden with appropriate conditions over the attributed graph. For presentation purposes, these conditions preventing such multi-valued attribution are implicitly assumed and not specified explicitly for every attributed graph.

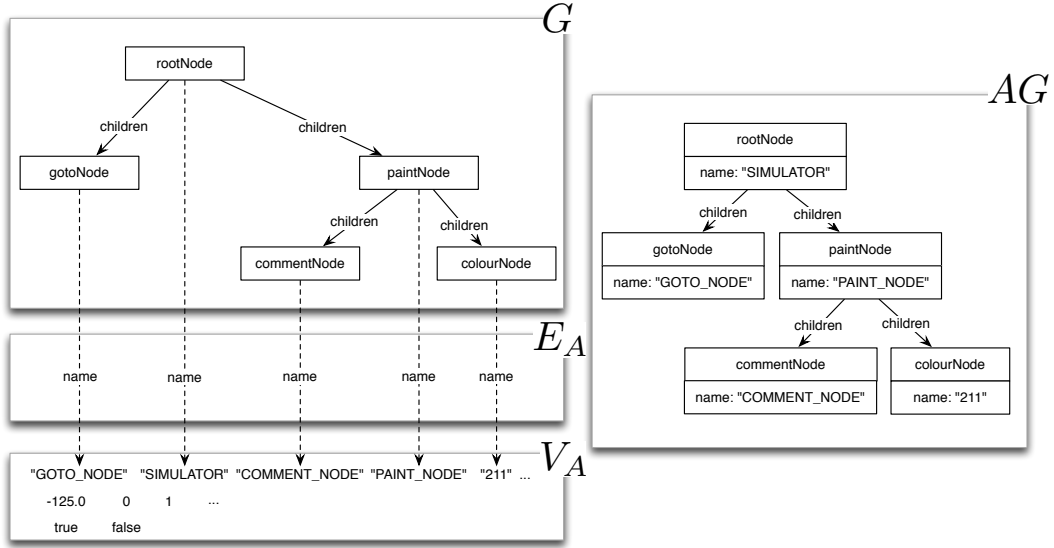


Figure 4.4: Formal and compact notation for attributed graphs

To type attributed graphs, the concept of a type graph is extended to attributed graphs in the following. To achieve this, a special kind of implementation of a given signature is required. A *final² algebra* implements a signature with carrier sets that consist of only one element, specifying the "type" of data nodes appropriately.

Definition 26 (Final Predicate Σ -Algebra).

Given a signature $\Sigma = (S, P)$, a *final predicate Σ -algebra* is an algebra Z with:

- carrier sets $Z_s = \{\bar{s}\}$, for each sort $s \in S$,
- predicates $p_Z : Z_{s_1} \times \cdots \times Z_{s_n} \rightarrow Z_{\text{Bool}}$ defined as $p_Z(\bar{s}_1, \dots, \bar{s}_n) := \overline{\text{Bool}}$ for each predicate symbol $p : (s_1, \dots, s_n) \rightarrow \text{Bool} \in P$.

For brevity, *final predicate Σ -algebras* will be referred to as *final algebras*.

² In category theory, final objects are dual to initial objects (cf. Def. 7) in the sense that there exist single arrows from every object in the category *to* the final (also called terminal) object. For algebras, the final algebra is, therefore, a "default" implementation of the algebraic signature such that every possible implementation (algebra) can be mapped into it.

Example 16 (*MocaTree as an attributed type graph*).

The following is a final algebra Z for the signature introduced in Ex. 14:

$$\begin{aligned} Z_{\text{Bool}} &= \{\text{EBoolean}\}, \quad Z_{\text{String}} = \{\text{EString}\}, \quad Z_{\text{Integer}} = \{\text{EInt}\}, \\ Z_{\text{Float}} &= \{\text{EDouble}\}, \quad \text{addSuffix}_Z(\text{EString}, \text{EString}, \text{EString}) = \text{EBoolean}, \dots \end{aligned}$$

This can now be used to define *MocaTree*, our tree metamodel from Chap. 3. *MocaTree* is depicted in Fig. 4.5 in the compact notation for attributed graphs. Compared to Fig. 3.3, note that in our current formalization, inheritance is flattened and multiplicities are not part of the type graph but must be specified as extra conditions.

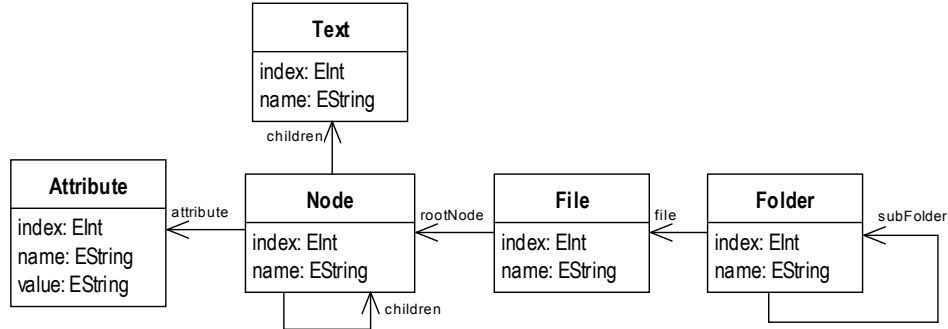


Figure 4.5: *MocaTree* as an attributed type graph

With the concept of final algebras, we are now ready to define metamodels and the conformance relation for attributed graphs analogously to Def. 4.

Definition 27 (*Typed Attributed Graphs and Typed Attributed Graph Morphisms*). Given a signature $\Sigma = (S, P)$, and a final algebra Z over Σ .

A *type attributed graph* ATG is a distinguished attributed graph

$\text{ATG} = (\text{TG}, V_Z, E_Z, s_Z, t_Z)$ over the final algebra Z .

A *typed attributed graph* is a pair $\widehat{\text{AG}} = (\text{AG}, \text{type})$ of an attributed graph AG together with an attributed graph morphism $\text{type} : \text{AG} \rightarrow \text{ATG}$.

Given typed attributed graphs $\widehat{\text{AG}} = (\text{AG}, \text{type})$ and $\widehat{\text{AG}}' = (\text{AG}', \text{type}')$,

a *typed attributed graph morphism* $f : \widehat{\text{AG}} \rightarrow \widehat{\text{AG}}'$ is an attributed graph morphism $f : \text{AG} \rightarrow \text{AG}'$ such that $\text{type}' \circ f = \text{type}$ (f is type preserving).

$\text{ATGraphs} = (\text{Ob}_{\text{ATGraphs}}, \text{Arr}_{\text{ATGraphs}}, \circ_{\text{ATGraphs}}, \text{id}_{\text{ATGraphs}})$ consists of:

- typed attributed graphs $\text{Ob}_{\text{ATGraphs}}$,
- typed attributed graph morphisms $\text{Arr}_{\text{ATGraphs}}$,
- $\circ_{\text{ATGraphs}} := \circ_{\text{AGraphs}}$
- $\text{id}_{\text{ATGraphs}} := \text{id}_{\text{AGraphs}}$

In analogy to Def. 4, $\mathcal{L}(\text{ATG}) := \{\text{AG} \mid \exists \text{type} : \text{AG} \rightarrow \text{ATG}\}$ denotes the set of all attributed graphs of type ATG .

Fact 9 (*ATGraphs is a category with pushouts along $\overline{\mathcal{M}}$ -morphisms in AGraphs*).

Proof. Arguments showing that **ATGraphs** is a category are analogous to **TGraphs** (cf. Fact. 3). The same pushout construction in **AGraphs** is valid in **ATGraphs** according to Fact A.19, Item 5 in [28]. \square

Example 17 (*Pattern Matching in ATGraphs*).

This example illustrates a typed attributed monomorphism $f : AG \rightarrow AG'$, that identifies (matches) an occurrence of the typed attributed graph AG in a *host graph* AG' .

Figure 4.6 depicts the match morphism f in a detailed notation and, for presentation purposes, with reduced AG, AG' , and type attributed graph ATG . The legend explains which arrows are used for the three components of the attributed morphisms $type : AG \rightarrow ATG$, $type' : AG' \rightarrow ATG$, and $f : AG \rightarrow AG'$, which are shown explicitly in the diagram.

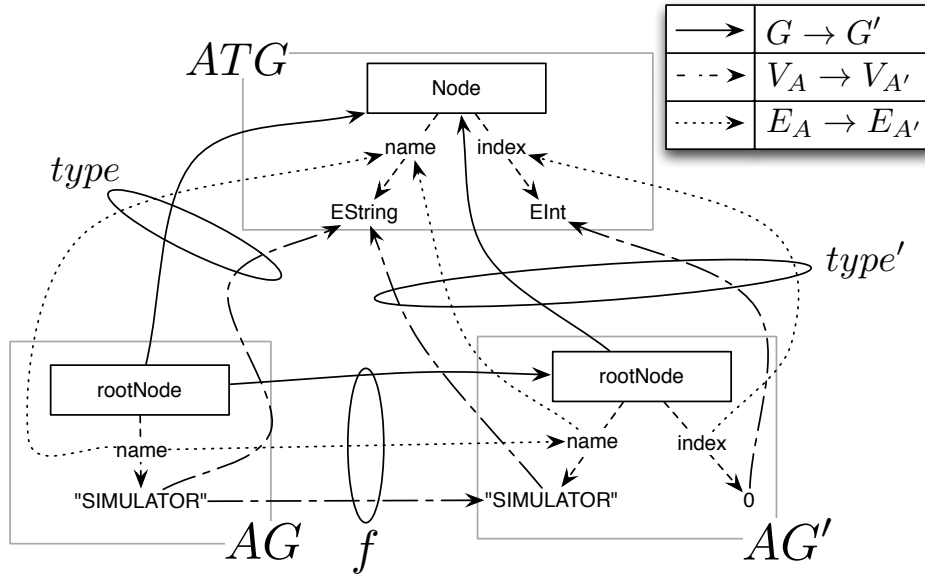


Figure 4.6: A match in AGraphs

Figure 4.7 depicts the same match morphism f in the compact notation for attributed graphs, now for complete AG, AG' , and ATG . Note that all elements in AG' with a pre-image in AG under f , are depicted with a bold frame and bold text. AG and AG' only consist of graph nodes of type **Node**.

As the diagram commutes, f is *type preserving* and can be represented as a diagram in **ATGraphs**, depicted in Fig. 4.8. Note that this combines two compact notations: one for representing attribute values as in-lined node entries, and one for representing the types of nodes as identifier:type, e.g., **rootNode:Node**.

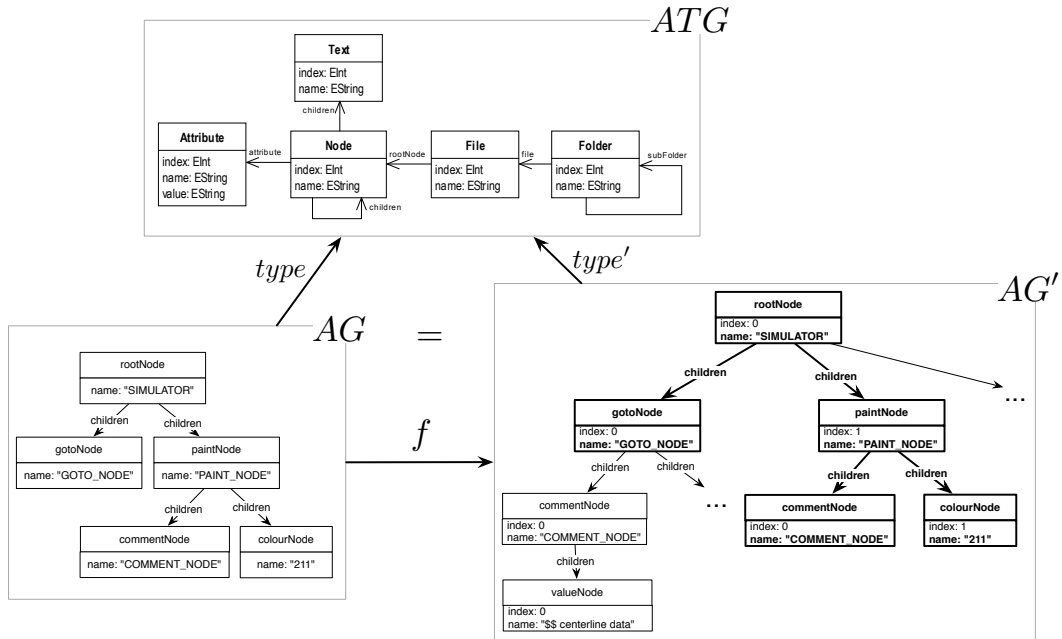


Figure 4.7: A match in AGraphs in compact notation

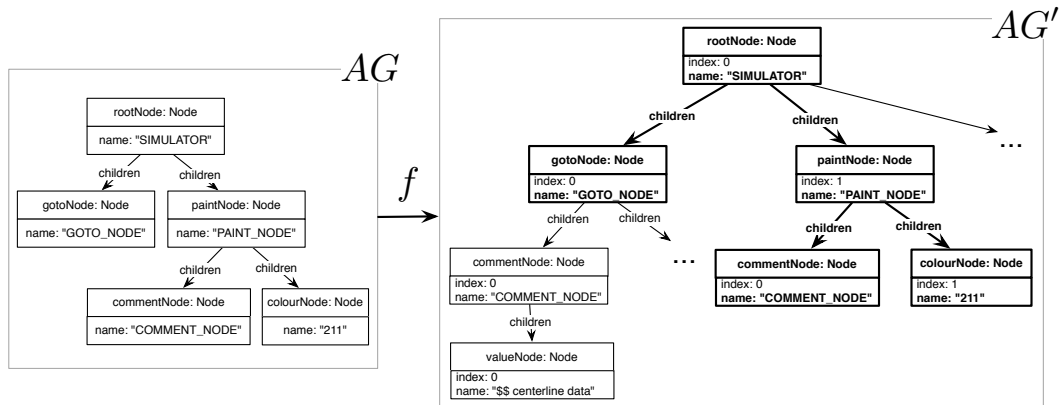


Figure 4.8: A match in ATGraphs in compact notation

4.1.2 Rule Application in **ATGraphs** with Attributes and Variables

After handling typing, the next step is to extend the concept of rules and derivations for typed attributed graphs. This is interesting as rules typically make use of *variables* to represent attribute values that are assigned a concrete value when determining a match.

The following formalizes how a given algebra can be extended by variables to yield a *term extension* of the algebra. This will be used as a suitable algebra for supplying the attribute values of the graphs in a rule. Note that arbitrarily nested terms are avoided in the definition.

Definition 28 (*Term Extension $A(X)$ of Algebra A with variables X*).

Given a signature $\Sigma = (S, P)$, an algebra A over Σ , and a family of sets of variables $X = \{X_s \mid s \in S\}$, where X_s are variables of sort s .

The *term extension* $A(X)$ of algebra A with variables X is an algebra over Σ with:

- carrier sets $A(X)_s := A_s \cup X_s$ for sorts $s \in S$
- predicates $p_{A(X)} : A(X)_{s_1} \times \cdots \times A(X)_{s_n} \rightarrow A(X)_{\text{Bool}}$,

$$p_{A(X)}(t_1, \dots, t_n) := \begin{cases} p_A(t_1, \dots, t_n) & \text{if } (t_1, \dots, t_n) \in A_{s_1} \times \cdots \times A_{s_n} \\ p_{A(X)}(t_1, \dots, t_n) & \text{otherwise} \end{cases}$$
for predicate symbols $p : (s_1, \dots, s_n) \rightarrow \text{Bool} \in P$,
- $A(X)_{\text{Bool}} := A_{\text{Bool}} \cup X_{\text{Bool}} \cup \{p_{A(X)}(t_1, \dots, t_n) \mid$

$$p : (s_1, \dots, s_n) \rightarrow \text{Bool} \in P,$$

$$(t_1, \dots, t_n) \in A(X)_{s_1} \times \cdots \times A(X)_{s_n}\}.$$

Example 18 (*A typed attributed graph over a term extension $A(X)$ of an algebra A*). —

A term extension with four variables for the algebra introduced in Ex. 14 is as follows:

$$\begin{aligned} \Sigma, A &: \text{ from Ex. 14} \\ X &= \{X_{\text{Bool}}, X_{\text{String}}, X_{\text{Integer}}, X_{\text{Float}}\} \\ X_{\text{Bool}} &= \emptyset \\ X_{\text{String}} &= \{\text{colourNode.name}\} \\ X_{\text{Integer}} &= \{\text{gotoNode.index}, \text{paintNode.index}, \text{paint.colour}\} \\ X_{\text{Float}} &= \emptyset \end{aligned}$$

The carrier sets are extended by the variables and, in the case of $A(X)_{\text{Bool}}$, by terms formed by applying the predicates in the algebra to the variables:

$$\begin{aligned}
 A(X) &= (\{A(X)_{\text{Bool}}, A(X)_{\text{String}}, A(X)_{\text{Integer}}, A(X)_{\text{Float}}\}, P_{A(X)}) \\
 A(X)_{\text{Bool}} &= A_{\text{Bool}} \cup \\
 &\quad \{\text{add}(\text{gotoNode.index}, 1, \text{paintNode.index}), \\
 &\quad \text{stringToNumber}(\text{colourNode.name}, \text{paint.colour}), \\
 &\quad \text{add}(0, 1, \text{gotoNode.index}), \dots\} \\
 A(X)_{\text{String}} &= A_{\text{String}} \cup X_{\text{String}} \\
 A(X)_{\text{Integer}} &= A_{\text{Integer}} \cup X_{\text{Integer}} \\
 A(X)_{\text{Float}} &= A_{\text{Float}}
 \end{aligned}$$

Figure 4.9 depicts a typed attributed graph AG , which is attributed over $A(X)$. Note, for example, the value of the `index` attribute of `gotoNode`, which is given as the variable `gotoNode.index`. The names of the variables in the term extension are already chosen systematically as `node.attribute` to increase readability. This convention will be used later to provide a more compact notation for typed attributed graphs over term extensions.

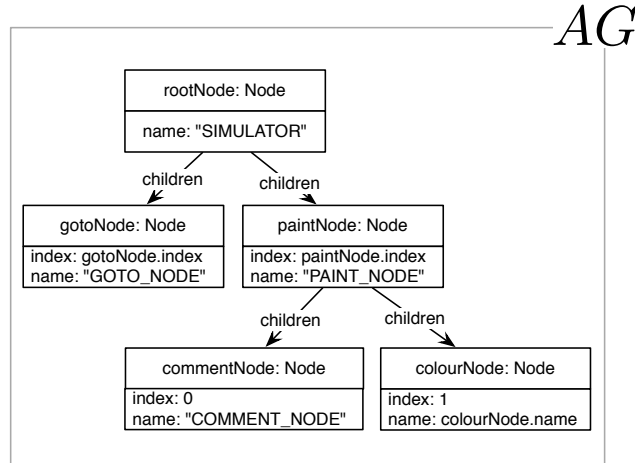


Figure 4.9: A typed attributed graph over a term extension $A(X)$ of an algebra A

The following definition formalizes rules and derivations with typed attributed graphs based on term extensions. The concept of a match morphism is extended appropriately by an assignment function that maps all variables and thus all terms in the term extension to values in the underlying algebra.

Definition 29 (*Attributed Rules, Attributed Graph Grammars, and Derivations*).

Given a signature $\Sigma = (S, P)$, a final algebra Z over Σ , an algebra A over Σ , and a term extension $A(X)$ of A with variables X .

An *attributed rule* is a typed attributed graph morphism $r : L \rightarrow R \in \overline{\mathcal{M}}$, where ATG is a type attributed graph over Z , and $L, R \in \mathcal{L}(ATG)$ are typed attributed

graphs over the same term extension $A(X)$. The data part of r , i.e., $r_{V_{A(X)}}$ is the identity $\text{id}_{V_{A(X)}}$ meaning that pushouts along r can be constructed.

An *attributed graph grammar* is a pair $GG = (ATG, \mathcal{R})$ of a type attributed graph ATG and a finite set \mathcal{R} of rules.

A *direct derivation* $AG \xRightarrow{r@m} AG'$ (or just $AG \xRightarrow{r} AG'$), for typed attributed graphs AG, AG' over the same algebra A used for the term extension $A(X)$, is given by a pushout in **ATGraphs**. The data part $m_{V_{A(X)}} : A(X) \rightarrow A$ of the match morphism $m : L \rightarrow AG$ restricted to A is completely determined by an assignment function $\xi : X \rightarrow A$, which fixes the values of all variables, while the graph part $m_G \in \mathcal{M}$ is a monomorphism (cf. Def. 8).

Derivation, $\mathcal{L}(GG, AG_\emptyset)$, and $\mathcal{L}(GG) = \mathcal{L}(GG, \emptyset)$ are defined analogously to Def. 8, where AG_\emptyset denotes the start typed attributed graph, and \emptyset the empty typed attributed graph.

Example 19 (*A direct derivation in ATGraphs*). —————

Figure 4.10 depicts a direct derivation $AG \xRightarrow{r@m} AG'$ for a rule $r : L \rightarrow R$ and match morphism $m : L \rightarrow G$ in **ATGraphs**. AG and AG' are constructed over the algebra A , and L, R are constructed over the term extension $A(X)$ of A with variables X as introduced in Ex. 18. All algebras are constructed over the same signature Σ .

The compact notation for denoting typing and attribution is used and the corresponding matches in each graph are emphasized as bold elements and bold text.

Two predicate terms are shown ① indicating how terms are built in the term extension. These predicate terms of type `Bool` are evaluated, i.e., mapped to either `false` or `true` in the underlying algebra by the data part of the match m . This evaluation is uniquely determined by the values assigned to all variables ② by the assignment function $\xi : X \rightarrow A$. In this case, the two predicate terms ① are evaluated to `true`. The rule r is applied to AG by building a pushout in **ATGraphs** according to the construction given in Fact. 8.

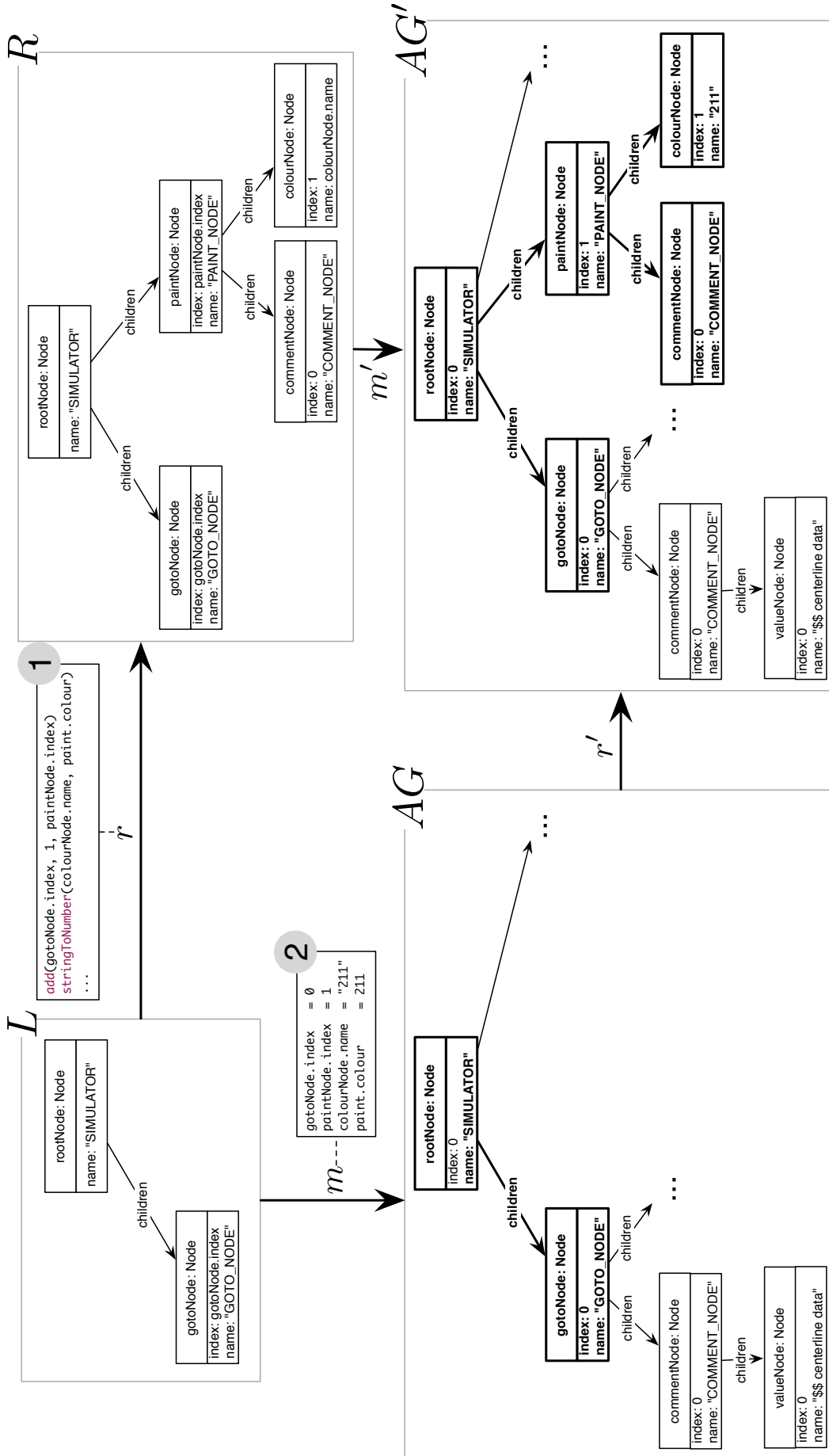


Figure 4.10: A direct derivation in ATGraphs

The rule applied in Fig. 4.10 is depicted in Fig. 4.11 in a compact notation for attributed rules. The notation is based on the previous compact notation for rules with typed graphs, i.e., green and ++ is used to denote elements in R without a pre-image in L (elements created by the rule). In addition, attribution in context elements is denoted by *assertions* of the form attribute == value, e.g., name == "SIMULATOR" in rootNode, while attribution in created nodes is denoted by *assignments* of the form attribute := value, e.g., index := 0 in colourNode.

A selection of predicate terms can be visualized in boxes connected to the graph nodes over whose attributes the predicates range. Each arrow connecting such a box with a graph node is also annotated with the name of the relevant attribute. For example, the box containing the predicate term `stringToNumber(colourNode.name, paint.colour)` is connected via an arrow annotated with `name` to `colourNode`. An important point to note here is that the naming convention of variables is used to simplify the diagram in the following manner: For every variable named `node.attribute`, an assertion (assignment) is assumed for the context (created) graph node `node`, if it exists, in the form `attribute == node.attribute` (`attribute := node.attribute`). This is why Fig. 4.11 is equivalent to the rule depicted in Fig. 4.10 even though, for instance, `gotoNode` in Fig. 4.11 does not explicitly contain the assertion `index == gotoNode.index`. If this is unwanted, then variables should be named to break this convention, e.g., `x`, `y`, `temp` or `paint.colour` as there exists no graph node with the identifier `paint` in the rule.

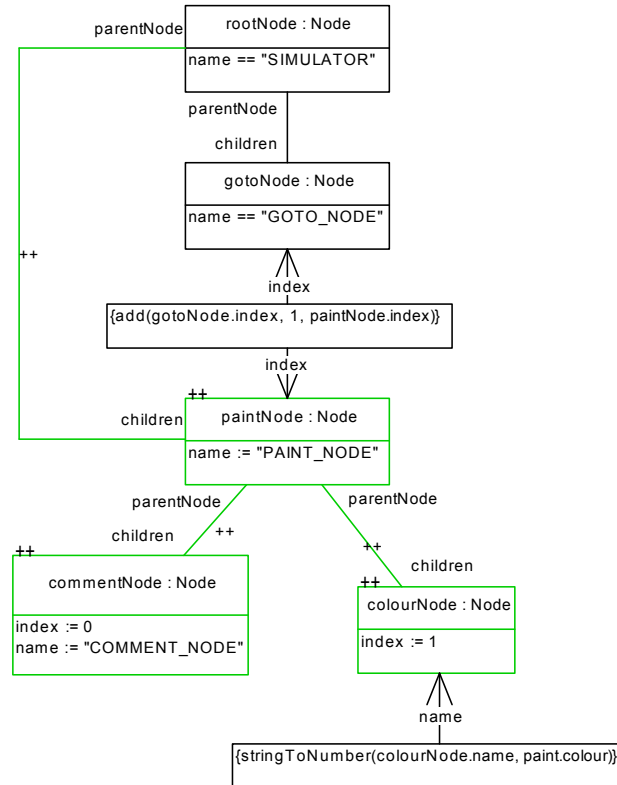


Figure 4.11: The rule applied in Fig. 4.10 in compact notation

4.1.3 Triples with Attributes, and Triple Rules with Attribute Conditions

We are now ready to lift all concepts necessary for attribution to *triples* of attributed graphs. The following states that this can be done analogously to Def. 13 and Def. 14, which lifted typed graphs to triples of typed graphs.

Definition 30 (*Typed Attributed Triple Graphs, Typed Attributed Triple Morphisms*). *Attributed triple graphs, attributed triple morphisms, and the category \mathbf{ATri} of typed attributed triple graphs and typed attributed triple morphisms, are defined analogously to Def. 13 and Def. 14, by replacing $\mathbf{TGraphs}$ with $\mathbf{ATGraphs}$.*

Fact 10 (*\mathbf{ATri} is a category with pushouts along $\overline{\mathcal{M}}$ -morphisms*).

Proof. Fact 4.18 in [28], already cited for lifting typed graphs to triples of typed graphs, is based on the categorical construction of *functor categories*, which is independent of the specific categories involved. $\mathbf{ATGraphs}$ together with the class of $\overline{\mathcal{M}}$ -morphisms can, therefore, be composed to triples in the same manner as $\mathbf{TGraphs}$, resulting in a well-defined category \mathbf{ATri} with pushouts constructed componentwise along $\overline{\mathcal{M}}$ -morphisms. The interested reader is referred to Def. A.36 and Fact A.37 in [28] for details on functor categories and pushout construction in functor categories. \square

Definition 31 (*Attributed Triple Rules, Attributed Triple Graph Grammar*). *Attributed triple rules and attributed triple graph grammars are defined analogously to Def. 15, by replacing $\mathbf{TGraphs}$ with $\mathbf{ATGraphs}$.*

As already hinted at in Ex. 19, a selection of predicate terms can be made and regarded as application conditions for the corresponding rule. For the rule to be applicable for a given match morphism, therefore, all the chosen predicate terms *must* evaluate to `true`. This concept of *attribute conditions* for TGG rules is formalized in the following.

Definition 32 (*Attributed Triple Rules with Attribute Conditions*).

Given a signature $\Sigma = (S, P)$, an algebra A over Σ , and a term extension $A(X)$ of A over a set of variables X .

An attributed triple rule with *attribute conditions* is a pair (r, \mathcal{C}_A) of an attributed triple rule $r : L \rightarrow R$ together with a set $\mathcal{C}_A \subseteq A(X)_{\text{Bool}}$ of Boolean terms.

A typed attributed triple morphism $m : L \rightarrow G$ satisfies a set $\mathcal{C}_A \subseteq A(X)_{\text{Bool}}$ of attribute conditions, denoted by $m \models \mathcal{C}_A$, if $m_{V_A}(t) = \text{true}$ for every $t \in \mathcal{C}_A$.

A direct derivation $AG \xRightarrow{r@m} AG'$ satisfies \mathcal{C}_A , denoted by $AG \xRightarrow{r@m} AG' \models \mathcal{C}_A$, if $m \models \mathcal{C}_A$.

Example 20 (*CLS to SimulatorLanguage Integration*).

To finalize this section on attribute manipulation in TGG rules, we now discuss the complete TGG specification required to realize the tree-to-model transformation introduced in Ex. 12, Fig. 3.7.

To connect our formalization to the actual concrete syntax used for TGG specifications in the model transformation tool eMoflon, Fig. 4.12 depicts the type attributed triple graph for the TGG. This overview of all correspondence elements and relevant elements in the source and target metamodels is often referred to as a *TGG schema*. The source elements *File* and *Node* form a small excerpt of the *MocaTree* metamodel (cf. Fig. 3.3) used to type CLS trees. As can be seen from the actual metamodels, our formalization is still incomplete: multiplicities correspond to extra conditions and are not part of the type graph, and *composite* references, denoted by a black diamond, will be handled as normal edges in this thesis. Finally, the concept of *bidirectional* references is also out-of-scope for our formalization, and references such as *file* \leftrightarrow *rootNode* are interpreted as two separate edges with no further conditions. The interested reader is referred to [90] for further details concerning composite / bidirectional references, and other modelling language features that are not particularly relevant for this thesis and are thus omitted from the formalization. A formalization of these features in the context of algebraic graph transformations is provided, for example, by [16, 95].

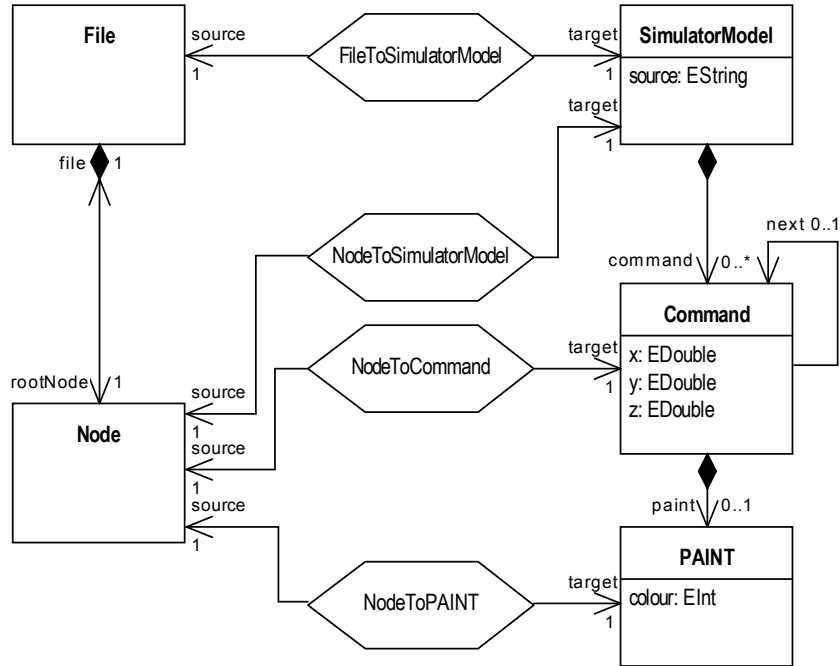


Figure 4.12: TGG schema for the running example

The target metamodel represents our simulator language, already introduced without attributes in Ex. 7, Fig. 2.18. Attributes have now been added to the model, e.g., the coordinate values of a Command and the identifier of the colour used by a PAINT. Finally, remember as always that, although the “links” connecting correspondence elements to source and target elements are visualized as normal source and target edges, they actually correspond to morphisms connecting the correspondence graph to its source and target graphs in the typed attributed triple graph. The concrete syntax is not intentionally chosen to be confusing and the reason for this particular mismatch with the formalization is that all TGG tools flatten triples to normal graphs for technical reasons. The interested reader is referred to [30] for a formal justification of this technique. The element-to-element mappings *can*, therefore, be thought of as special edges, especially for end-users who are not primarily interested in the formalization.

The first rule is the axiom `FileToSimulatorRule` depicted in Fig. 4.13. It creates a file, its root node, and a corresponding simulator model. Concerning attribute manipulation, the root node’s name is set to “SIMULATOR”, while a predicate term is used to express an attribute condition over the name of the file and the source of the simulator model. Demanding that `addSuffix(root.source, “.cls”, file.name)` be evaluated to true by any match restricts possible values for both attributes to pairs of values for which the attribute condition (implemented as its name suggests) is fulfilled. Such pairs include for instance (“v0”, “v0.cls”) and exclude (“v1”, “test.txt”). In this manner, the match is free to assign any *consistent* pair of attribute values.

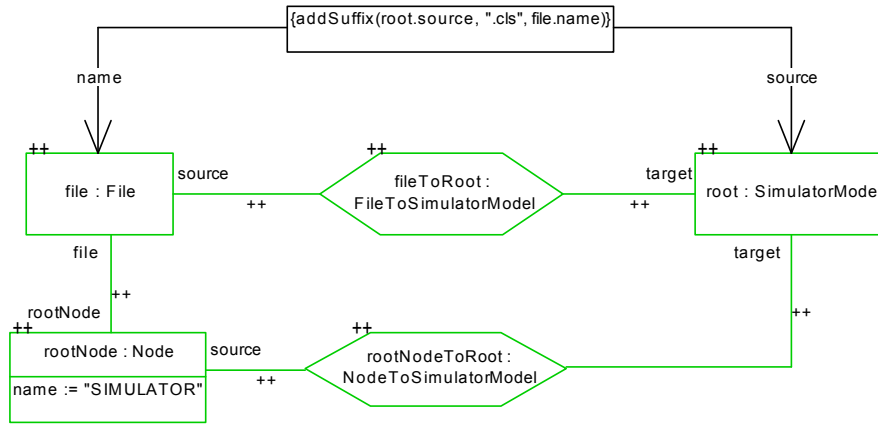


Figure 4.13: FileToSimulatorRule

As the folder containing the CLS file has absolutely no relevance for the corresponding simulator model, the ignore rule `IgnoreFolderRule` depicted in Fig. 4.14 is used to express this fact. This basically means that the consistency relation is not affected by placing the file in a folder. Note that we also do not care what the name of the folder is.

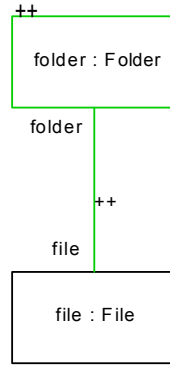


Figure 4.14: IgnoreFolderRule

The next rule `FirstGotoRule` creates a `Command` in the simulator model together with the corresponding source tree structure. Assertions and assignments are used to demand and set fixed attribute values in the tree, respectively. The argument values are demanded to be consistent with the coordinate values in the command using suitable attribute conditions. As the name of the predicate `stringToNumber` implies, it evaluates to true for pairs of numbers and their textual representation such as `("211", 211)`. As this rule does not demand any other existing `Command` in the simulator model, it can be used to start the sequence of commands, explaining the chosen name of the rule (`First...`).

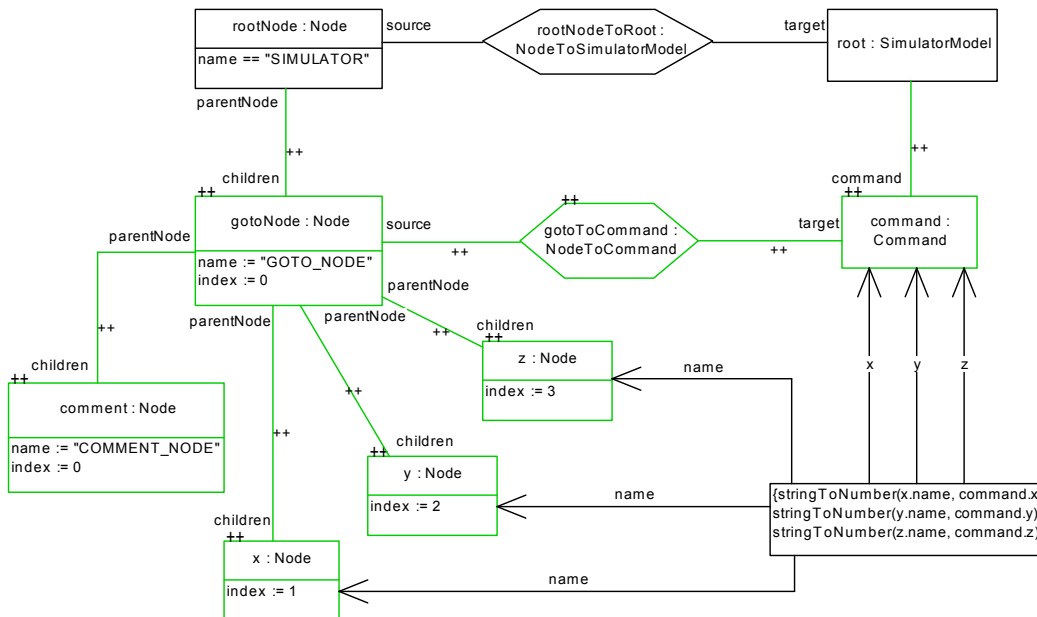


Figure 4.15: FirstGotoRule

As comments in CLS files, parsed if present as the single child of the comment node in `FirstGotoRule`, are also irrelevant for the simulator model, the ignore rule `IgnoreCommentRule` depicted in Fig. 4.16 is used to express this.

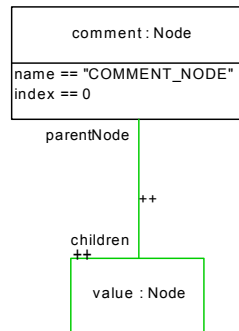


Figure 4.16: IgnoreCommentRule

The remaining three rules in the TGG are used to build up sequences of connected Commands. After the initial Command is created with FirstGotoRule, the sequence of Commands can be extended by appending:

1. A new Command to a preceding Command (handled by GotoGotoRule),
2. A new PAINT to a preceding Command (handled by GotoPaintRule), and
3. A new Command to a preceding PAINT (handled by PaintGotoRule).

This means that a sequence of multiple PAINTS is forbidden in our (current) simulator language, i.e., a rule for appending a PAINT directly after a PAINT is not specified.

GotoGotoRule depicted in Fig. 4.17 is similar to FirstGotoRule but demands a preceding GOTO_NODE in the tree and a preceding Command in the simulator model. Although this can be achieved with a single next edge in the simulator model, the predicate `add(prevGotoNode.index, 1, gotoNode.index)` is necessary to ensure that the correct `prevGotoNode` is matched in the tree.

To handle a PAINT Command, GotoPaintRule depicted in Fig. 4.18 creates a corresponding paint node in the CLS tree with a comment node and a second child node representing the colour code of the paint command as a string. The attribute condition `stringToNumber(colourNode.name, paint.colour)` ensures that the value of the colour attribute of the paint command is consistent with the name of the colour node. Note that the created PAINT belongs to the previous Command, i.e., is connected via a paint and not a next edge.

To append a Command after a previous Command that has a PAINT, the corresponding tree structure is demanded as context in PaintGotoRule depicted in Fig. 4.19, i.e., a preceding goto node followed directly by its paint node. If such a structure exists, then a new goto node can be appended analogously to GotoGotoRule.

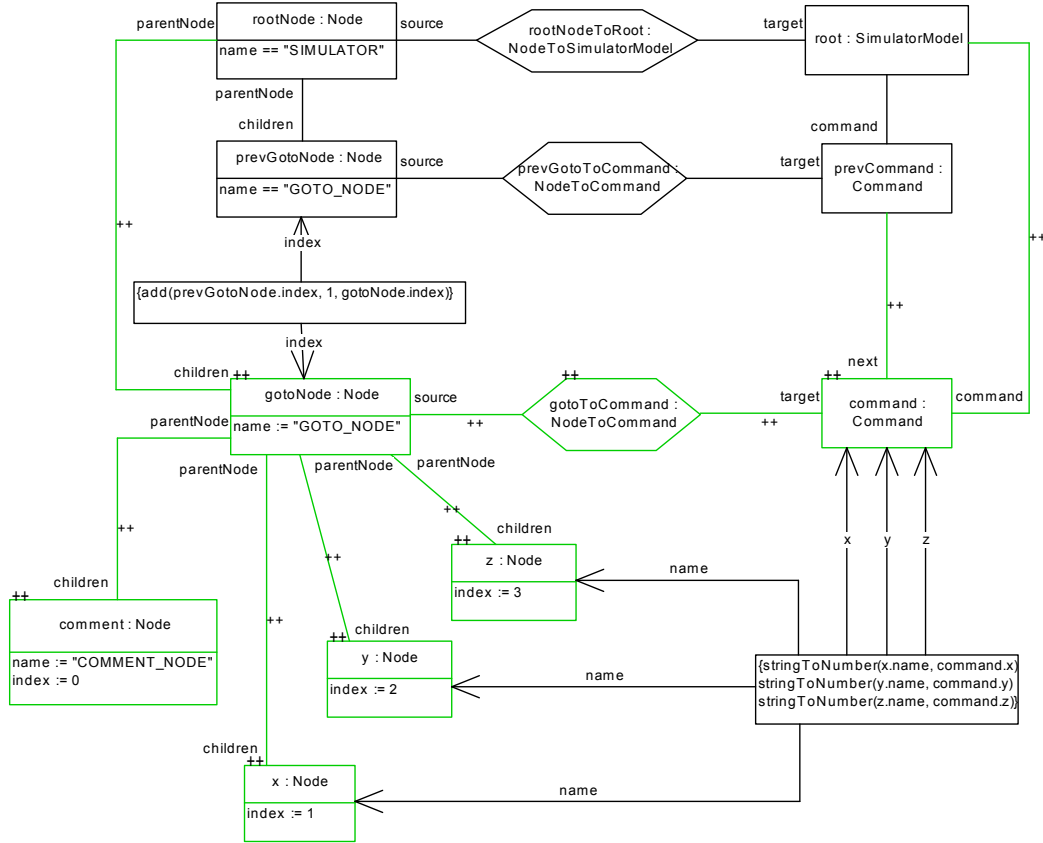


Figure 4.17: GotoGotoRule

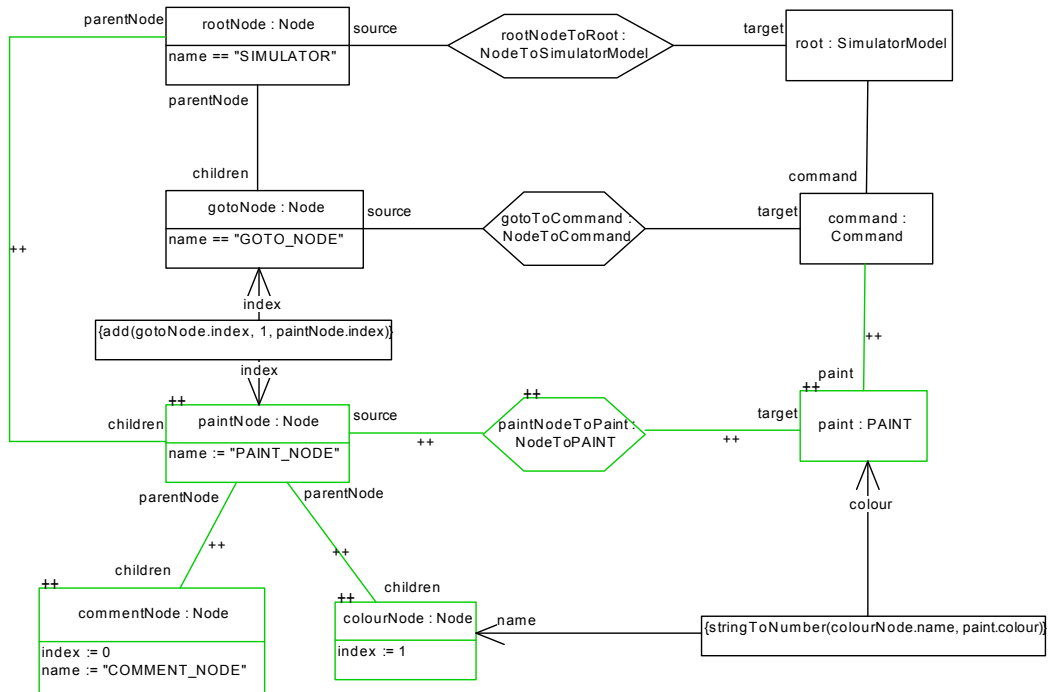


Figure 4.18: GotoPaintRule

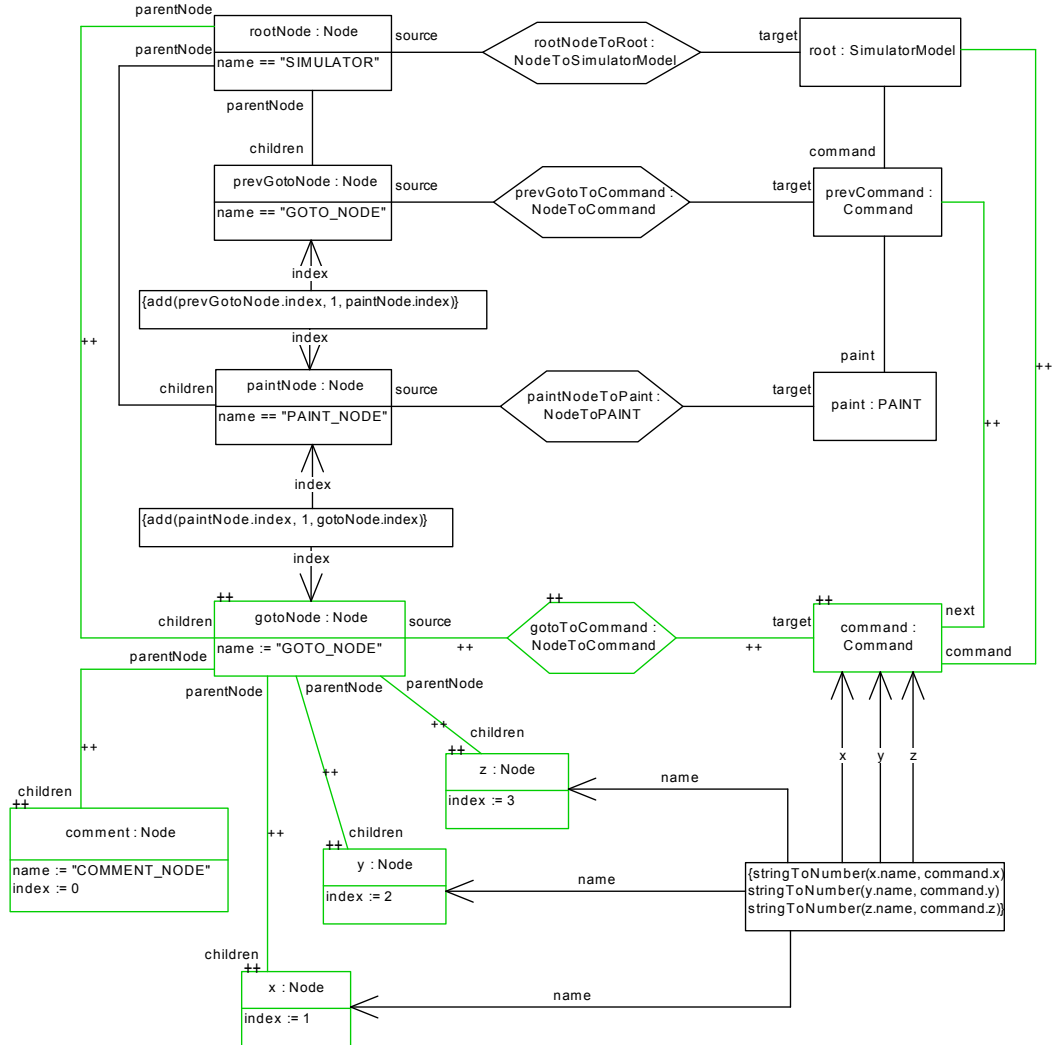


Figure 4.19: PaintGotoRule

4.2 ADDITIONAL STRUCTURAL CONSTRAINTS WITH DYNAMIC CONDITIONS

When the source, target or both metamodels for a TGG cannot be extended or changed, it can be difficult or even impossible to express certain structural conditions as graph patterns, whose size is determined *statically* at rule specification time. To handle these cases, there exist various approaches including (i) providing a further *path expression* language for specifying certain sequences of edges [87], and (ii) *recursive pattern invocation*, with which a pattern can be unfolded as often as necessary [59]. These approaches and other techniques allow the size of graph patterns in rules to be determined *dynamically* at rule application time for a specific match in a concrete host graph.

To investigate how such extensions can be integrated in TGG rules, the concept of *dynamic conditions* is introduced in this section. Dynamic conditions provide a simple and high-level interface for integrating additional application conditions in a rule, which are checked dynamically at rule application time.

To be compatible with path expressions, recursive pattern invocation, and other (future) ideas for extending graph patterns dynamically, a dynamic condition over a graph D demands an *interface graph* I , which is embedded in D , via an *interface morphism* $i : I \rightarrow D$. This defines the part of the graph that is required to evaluate the dynamic condition, i.e., forms the *interface* of the dynamic condition.

For a path expression, the interface could consist of start and end nodes of the path, whereas for recursive pattern invocation, all nodes and edges that serve as “anchors” for the pattern unfolding could form the interface.

At rule application time, the implementation of a dynamic condition must take a match $m : D \rightarrow G$ and check if the condition holds for m . This is formalized via a function β , that takes m and decides if the interface I can be extended to a certain graph C (the conclusion), i.e., attempts to construct $\beta(m) : I \rightarrow C$.

For a path expression, the conclusion C would be the dynamically determined path, for recursive pattern invocation, C could be the dynamically determined match for the unfolded pattern. If the conclusion cannot be constructed then the dynamic condition fails, blocking rule application. The following definition formalizes this idea.

Definition 33 (*Dynamic Condition*).

Given a category \mathbf{C} , a *dynamic condition* dc over an object D is a pair (i, β) , where $i : I \rightarrow D$ is an arrow referred to as the *interface morphism* that embeds the *interface object* I in D , and β is a function that maps morphisms $m : D \rightarrow G$ to morphisms $\beta(m) : I \rightarrow C$, which extend the interface object I to the *conclusion* C .

A morphism $m : D \rightarrow G$ satisfies a dynamic condition $dc = (i, \beta)$, denoted by $m \models dc$, if there exists a $q : C \rightarrow G$ with $\beta(m) : I \rightarrow C$, for which Fig. 4.20 commutes.

Given a TGG rule $r : L \rightarrow R$ a dynamic condition dc over L is referred to as a *dynamic precondition*, while a dynamic condition dc over R is referred to as a *dynamic postcondition*.

A derivation $G \xRightarrow{r@m,m'} G'$ satisfies a dynamic pre-condition dc , denoted by $G \xRightarrow{r@m,m'} G' \models dc$, if $m \models dc$.

A derivation $G \xRightarrow{r@m,m'} G'$ satisfies a dynamic post-condition dc , denoted by $G \xRightarrow{r@m,m'} G' \models dc$, if $m' \models dc$.

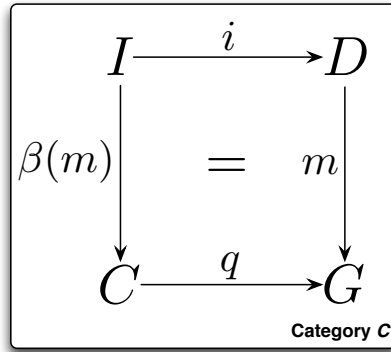


Figure 4.20: Satisfaction of dynamic conditions

Example 21 (*Handling modal PAINT commands with dynamic conditions*). —————

To provide further motivation for this idea based on a concrete example, let us extend our running example by adding an additional requirement inspired by the actual transformation implemented in the context of the CME project introduced in Chap. 1. We shall assume that PAINT commands in CLS files are *modal*,³ in the sense that the colour argument does not have to be specified if the former value would just be repeated.

This means that the following CLS snippet:

```
GOTO/-125.0,20.0,30.0
PAINT/COLOR,211
GOTO/-125.0,-23.333,30.0
PAINT/COLOR,211
...
```

is identical to:

```
GOTO/-125.0,20.0,30.0
PAINT/COLOR,211
GOTO/-125.0,-23.333,30.0
PAINT/
...
```

³ In the CME project, the argument values of G-commands in MPF files are, for instance, modal.

Note that both snippets are valid (consistent) and allowed, i.e., as explained in detail in Chap. 1, sometimes the modal version is preferred, sometimes not. To address this new requirement, an additional rule `GotoModalPaintRule` is required to handle the case where the colour code is omitted in the CLS file, indicating that the value of `paint.colour` must be consistent with the value of the last paint node in the tree that had a non-trivial⁴ colour node.

It is surprisingly difficult to express this directly as the left hand side of a rule, i.e., as part of an attributed typed graph, because the exact required context structure *depends on the host graph* and cannot be explicitly stated statically (at rule specification time and thus independent of the match morphism).

One can always avoid such problems by extending one or both of the metamodels and adding a pre- and postprocessing step, or in some cases, by (mis)using correspondence links to “remember” certain elements required in other rules. The former solution, however, is often impossible, unwanted due to adverse effects of pre- and postprocessing on derived synchronizers, and potentially inefficient. The latter solution is often a “hack” that reduces readability of the rules (correspondences are now technical crutches and no longer represent connections of elements in the metamodels), and is also potentially inefficient with respect to memory usage (leads to large correspondence models).

In some cases, it is also possible to rephrase such a condition so it can be specified as a negative application condition that is independent of the match morphism. We shall see in Chap. 5, however, that such conditions complicate the derivation of synchronizers and are to be avoided.

Figure 4.21 depicts the new rule `GotoModalPaintRule` that makes use of a dynamic condition to handle modal paint commands. In practice, the interface graph for dynamic conditions is restricted to two nodes, referred to as the *source* and *target* node of the interface graph. In the concrete visual syntax used in Fig. 4.21, therefore, the interface graph is depicted as a “virtual edge”, i.e., a dashed arrow going from the source node (`paintNode`) to the target node (`lastNonTrivialColourNode`). The target node is depicted with a bold frame for additional emphasis.

The colour node `colourNode` created for the modal paint node `paintNode` is trivial, i.e., its name is set to the empty string `""`, and the colour of the created paint command in the simulator model is set to be consistent with `lastNonTrivialColourNode`.

As a final shorthand notation, note that certain simple attribute conditions⁵ can be in-lined in nodes, such as `name != ""` in `lastNonTrivialColourNode`. This notation is thus equivalent to the attribute condition `noteq(lastNonTrivialColourNode.name, "")` using the predicate symbol `noteq` for `!=`.

An important question is if modal paint commands can be handled with `GotoPaintRule`. This would be undesirable as the colour code `""` does not correspond to any colour. This is an example for a case where it would be wrong to have a degree of freedom, i.e., allowing both rules to be applicable.

⁴ The label (name) of the colour node is not the empty string, i.e., `""`

⁵ `!=` for strings and integers, `<`, `≤`, `>`, `≥` for integers.

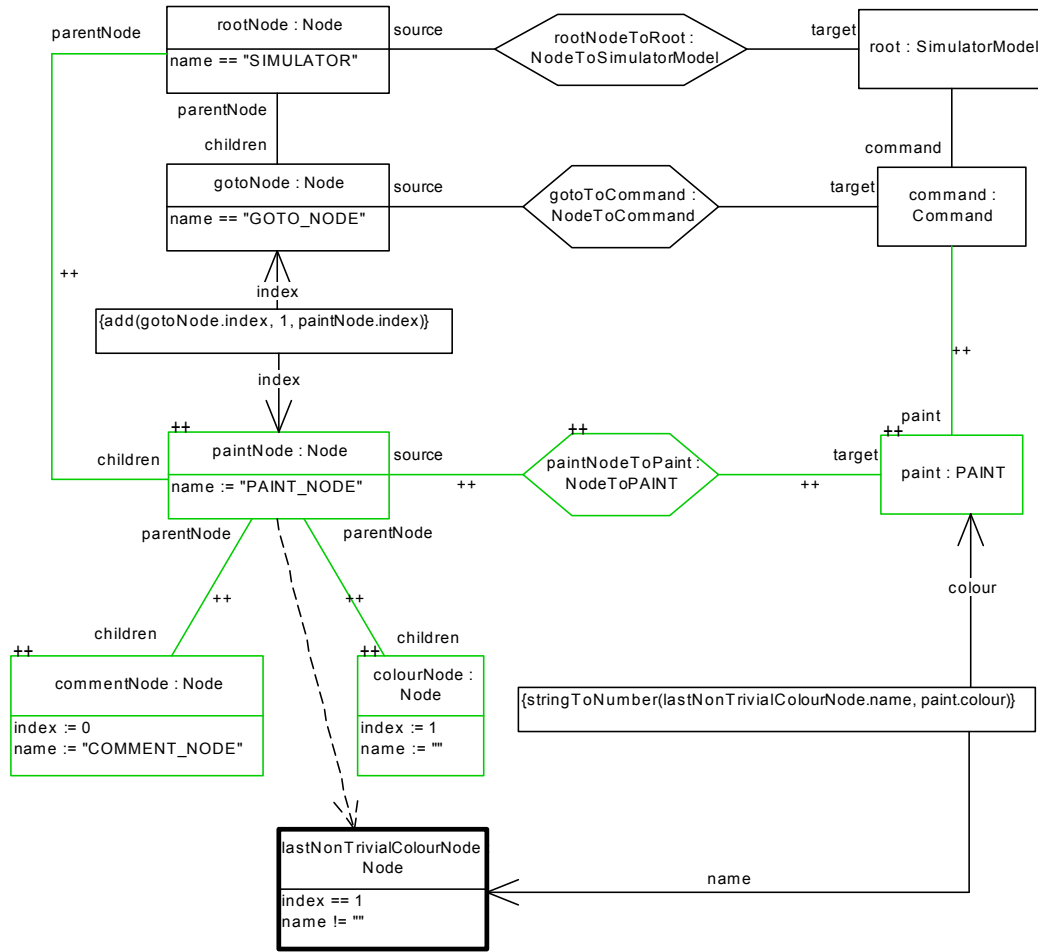


Figure 4.21: GotoModalPaintRule for handling modal PAINT commands

Considering `GotoPaintRule` in Fig. 4.18, however, it is clear that modal paint commands cannot be handled as `colourNode.name` would be assigned the value "" (empty string), for which the attribute condition `stringToNumber(colourNode.name, paint.colour)` clearly evaluates to false, conveniently blocking the application of the rule. This ensures that modal paint commands can only be handled by `GotoModalPaintRule` and only if the dynamic condition is satisfied, i.e., only if a previous non-trivial colour node can actually be found.

This means that the following CLS fragment is, for instance, invalid as the modal paint has no previous value to refer to:

```
// Start of file
GOTO/-125.0,20.0,30.0
PAINT/
GOTO/-125.0,-23.333,30.0
...
```

Figure 4.22 depicts a direct derivation with the new rule `GotoModalPaintRule`, showing how the dynamic condition is satisfied for a concrete host graph AG . In the graphs AG , AG' , and C , the images of the match morphisms m , m' , and $\beta(m')$ are indicated by depicting the relevant elements with a bold frame and by using bold text. For presentation purposes, all correspondence links are simplified and attribute conditions are omitted to keep the diagram as simple as possible. All missing details of the rule `GotoModalPaintRule` can be inferred from Fig. 4.21.

In AG , the third goto node in the CLS tree is identified by m , and according to `GotoModalPaintRule`, a modal paint node is to be added directly after it. For the rule to be applicable however, the dynamic condition $(i : I \rightarrow R, \beta)$, must be satisfied by m' , i.e., is in this case a *post-condition* for the rule. Note that the dynamic condition could also have been formulated as a *pre-condition* for the rule but, in some cases, it is easier in practice to implement β for the corresponding post-condition. For this reason, the formalization is chosen to be more general in this regard.⁶

In this case, the interface graph I consists of two nodes which are mapped by the interface morphism i to the newly created paint node `paintNode`, and the last non-trivial colour node `lastNonTrivialColourNode`. As required, $\beta(m')$ extends I to a path of command nodes in the CLS tree, leading to the last paint node with a non-trivial colour node.

For the concrete host graph AG , this path consists of two goto nodes, `gotoNode` and 2, as well as the paint node 1. Note that C and $q : C \rightarrow AG'$ depend on m' and must be determined dynamically when attempting to apply the rule. For the next paint command in AG' , for instance, β would have to skip modal paint commands such as the newly created `paintNode` during the search for the last non-trivial paint node (which would be the same node `lastNonTrivialColourNode` as in Fig. 4.22).

⁶ This does not make such a big difference for the TGG compiler as all source (target) dynamic post-conditions become pre-conditions anyway in forward (backward) rules (cf. Chap. 5, 7).

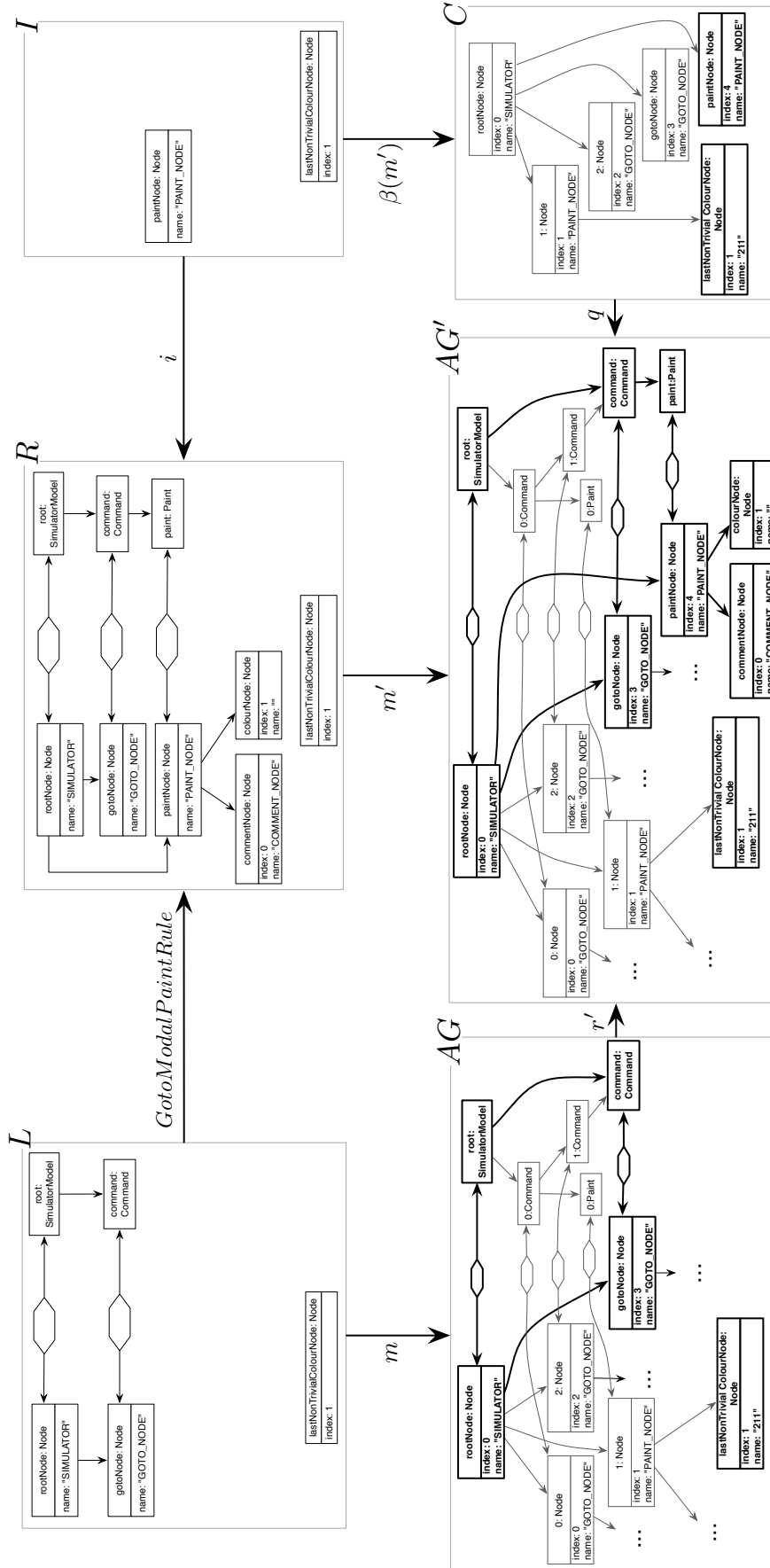


Figure 4.22: Direct derivation with GotoModalPaintRule

4.3 MODULARIZING TGGs WITH RULE REFINEMENT

With the preceding sections in this chapter, we have now introduced all relevant TGG language features required and established for this thesis. Recall that attribute conditions and dynamic conditions are especially useful for *vertical* TGGs, where attribute values and complex structural relationships on the lower abstraction level typically correspond to types and simple edges on the higher level of abstraction, respectively. Using this set of language features, the CME project, described in detail in Chap. 1, was implemented with about 46 TGG rules.

Although we have addressed *expressiveness*, readers with a software engineering background must be feeling deeply disturbed by the TGG introduced in Ex. 20. The problem is that almost all rules have an *identical recurring* substructure repeated in each rule with minimal changes. For example, `FirstGotoRule`, `GotoGotoRule`, and `PaintGotoRule` all create a `Command` and the corresponding goto tree structure *in exactly the same way*. For realistic TGGs with more than 30 rules, this leads to a maintenance nightmare when such identical substructures must be consistently changed in all relevant rules.

To avoid such *pattern duplication*, a means of *reusing* common patterns in multiple rules is required. In addition, the reuse mechanism must be flexible enough to handle cases where the common pattern is not exactly the same but is only slightly changed.

In the following, therefore, a novel modularity concept will be established for TGGs, based on the ideas and concepts introduced in [10]. Figure 4.23 depicts a roadmap for the following sections covering *rule refinement*. The challenge of avoiding pattern duplication in rules is addressed by providing a concise pattern language with which higher-order transformations (using rule patterns to transform rule patterns) can be specified, thus enabling a flexible composition and reuse of common (sub)patterns in rules.

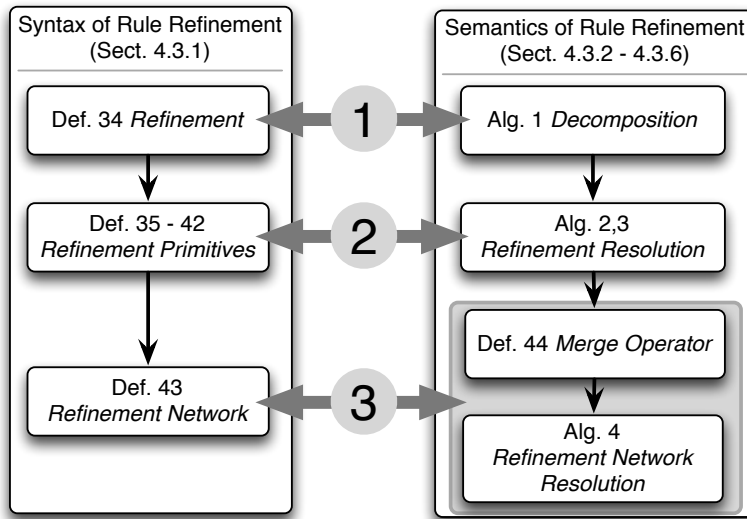


Figure 4.23: Roadmap for the following discussion on rule refinement.

The basic idea is to establish a suitable and compact language for describing *rule refinements*, i.e., the changes required to produce a new rule from a set of basic rules. We shall first of all define the syntax of rule refinement in Sect. 4.3.1, which is chosen to fit the existing TGG syntax for rules.

This consists of: ① the general structure of a *rule refinement* (Def. 34), ② a set of *rule refinement primitives* (Def. 35 – 42) corresponding to atomic refinement operations such as deleting a single edge, or adding a single node, and ③ the syntactical structure of a *refinement network*, a hierarchy of refinements used to merge and compose multiple refinements.

The semantics of rule refinement is discussed next in Sect. 4.3.2 – Sect. 4.3.6, starting by ① decomposing a rule refinement into a set of refinement primitives. This decomposition process reduces the semantics of an arbitrary refinement pattern to the semantics of a fixed set of refinement primitives. Each refinement primitive is then *resolved* ② by sorting the primitives and executing a corresponding model transformation, e.g., deleting an edge from a rule for a *DeleteEdge* refinement primitive. Finally, the resolution process is extended ③ to cover refinement networks. This involves a *merge operator* required to deal with multiple refinement, i.e., refining more than one basis rule.

4.3.1 Syntax of Refinements: Refinement Structure and Refinement Primitives

A *refinement* consists syntactically of two triple rules connected in such a manner that it is clear (according to Def. 34 – 42) which elements are to be deleted, replaced, or newly created.

Definition 34 (*Refinement*).

Let Σ be a signature and A an algebra over Σ .

A *refinement* $\Delta(r^*, r)$ consists of two attributed triple rules $r^* : L^* \rightarrow R^*$ and $r : L \rightarrow R$, over the same predicate term extension $A(\Sigma)$, connected by attributed triple morphisms $\delta_L, \delta_{L^*}, \delta_R, \delta_{R^*}$, typed attributed triple graphs Δ_L, Δ_R , and a typed triple morphism $\delta : \Delta_L \rightarrow \Delta_R$, such that the diagram depicted in Fig. 4.24 commutes.

The *refining* rule r^* is said to *refine* its *basis* rule r .

Note that $\delta_L, \delta_{L^*}, \delta_R, \delta_{R^*}$ are *not* typed.

In the following, we shall take a compositional approach and define a series of refinement *primitives*, each representing an executable atomic modification to the basis rule. Complex refinements are composed by combining these primitives.

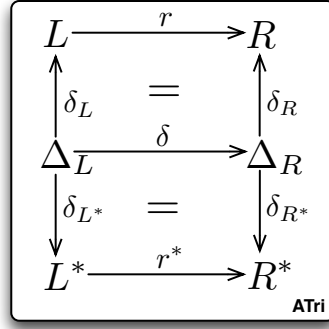
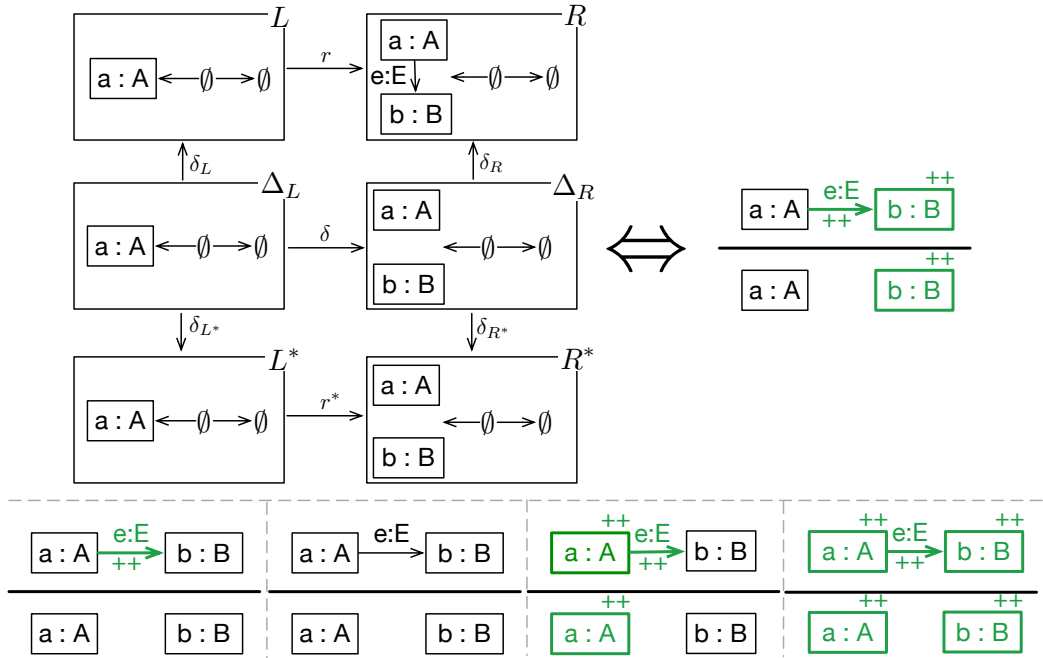


Figure 4.24: Syntax of refinements

Definition 35 (*DeleteEdge*).

A *DeleteEdge* source refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the five diagrams depicted in Fig. 4.25.

DeleteEdge target refinements are defined analogously, i.e., with non-trivial target components.


 Figure 4.25: *DeleteEdge* source refinements

The first *DeleteEdge* diagram is depicted in both a detailed syntax to the left, and an equivalent compact syntax to the right. In the detailed syntax, elements in the attributed typed graphs are denoted by `label:type` giving a label for the element and its type.

The graph morphisms $\delta_{L_S}, \delta_{L_S^*}, \delta_{R_S}, \delta_{R_S^*}$, depicted as arrows, are given implicitly by requiring unique element labels in each graph and mapping equally labelled nodes (not necessarily of the same type) to each other, and equally labelled edges to each other.

In the compact syntax, only non-trivial graphs are shown (in this case only the source components). The basis rule is placed above the black horizontal line, while the refining rule is placed below. Analogously to the compact syntax for rules, elements in $R_S \setminus L_S$ are annotated with a “++” markup⁷ to differentiate them from elements in L_S . This allows for a compact notation, which is used for all other cases. Fig. 4.25 depicts in sum five different variants of the *DeleteEdge* refinement primitive.

Definition 36 (*CreateEdge*).

A *CreateEdge* source refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the five diagrams depicted in Fig. 4.25 but with the roles of L/L^* and R/R^* exchanged.

CreateEdge target refinements are defined analogously.

Example 22 (*DeleteEdge* and *CreateEdge* source refinements).

To demonstrate *DeleteEdge* and *CreateEdge* source refinements, Fig. 4.26 depicts refinements based on the TGG rule *IgnoreFolderRule* from our running example. The refinement to the left is a *DeleteEdge* source refinement, indicating that the folder edge between file and folder is to be deleted, while the refinement to the right is a *CreateEdge* source refinement, indicating that the same edge is to be created.

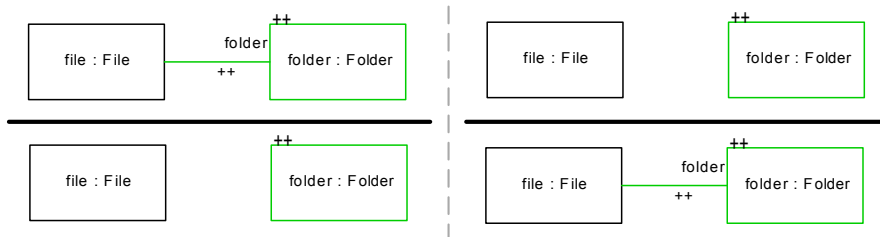


Figure 4.26: *DeleteEdge* and *CreateEdge* source refinements

⁷ Additionally emphasized by depicting them in green instead of black.

Definition 37 (*ReplaceNode*).

A *ReplaceNode* source refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the four diagrams depicted in Fig. 4.27.

As indicated in the diagrams with “...” representing any attribute values, *node* in this context denotes the graph node a and all incident attribute edges. A graph node is, therefore, replaced together with all its attribute edges.

ReplaceNode target refinements are defined analogously.

Note that the type of the replaced node can be changed in general, i.e., the graph morphisms $\delta_{L_S}, \delta_{L_S}^*, \delta_{R_S}, \delta_{R_S}^*$ are not necessarily type preserving (cf. Def. 34).

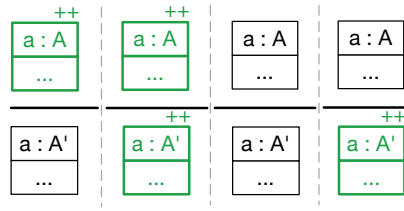


Figure 4.27: *ReplaceNode* source refinements

Definition 38 (*CreateNode*).

A *CreateNode* source refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the two diagrams depicted in Fig. 4.28.

As indicated in the diagrams with “...” representing any attribute values, *node* in this context denotes the graph node a and all incident attribute edges. A graph node is, therefore, created together with all its attribute edges.

CreateNode target refinements are defined analogously.

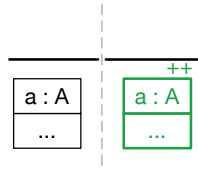


Figure 4.28: *CreateNode* source refinements

Example 23 (*ReplaceNode* and *CreateNode* source refinements).

Figure 4.29 depicts a concrete example of a *ReplaceNode* and a *CreateNode* source refinement to the left and to the right, respectively. Note that the *ReplaceNode* source refinement also adds an attribute assignment together with *colourNode*.

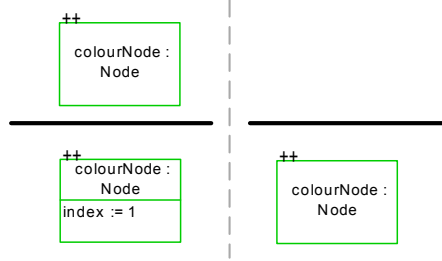


Figure 4.29: *ReplaceNode* and *CreateNode* source refinements

Definition 39 (*DeleteCorr*).

A *DeleteCorr* refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the five diagrams in Fig. 4.30.

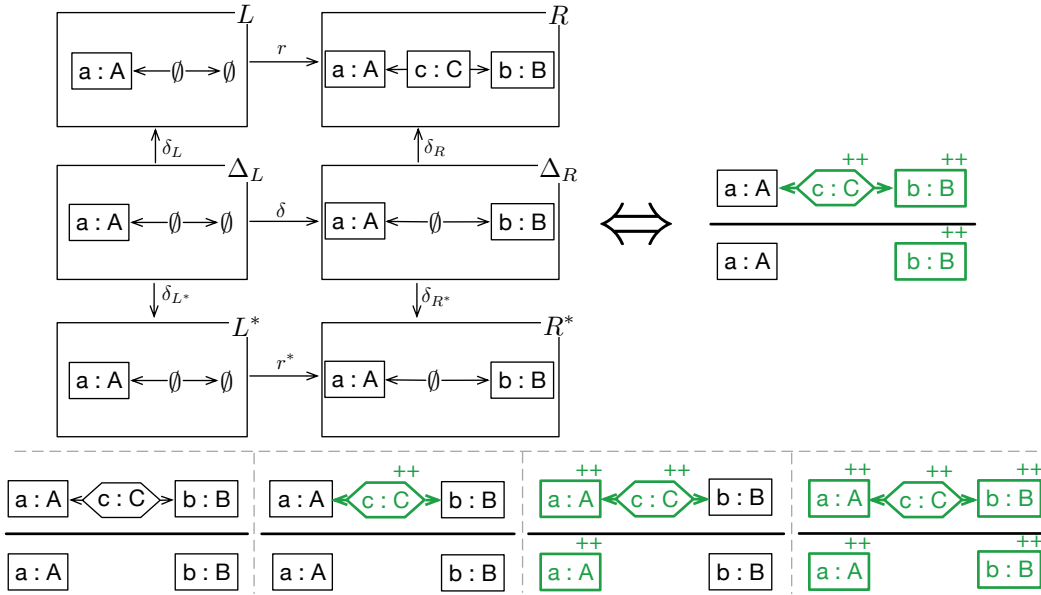


Figure 4.30: *DeleteCorr* refinements

Note that, analogously to Fig. 4.25, the first *DeleteCorr* refinement is depicted in a detailed formal syntax to the left and a compact syntax to the right. For presentation purposes, the latter is used for the rest of the refinements.

Definition 40 (*CreateCorr*).

A *CreateCorr* refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the five diagrams in Fig. 4.30 but with the roles of L/L^* and R/R^* exchanged.

Example 24 (*DeleteCorr refinement*).

Figure 4.31 depicts a concrete example of a *DeleteCorr* refinement. Analogously to edges, the correspondence element `rootNodeToRoot` is implicitly deleted in r^* by simply not repeating it. A *CreateCorr* refinement can be constructed by reverting the roles of r and r^* in the example.

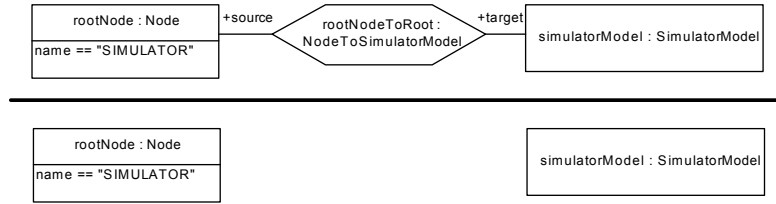


Figure 4.31: *DeleteCorr* refinement

Definition 41 (*AddCondition*).

An *AddCondition* refinement is a refinement $\Delta(r^*, r)$, with an application condition, dynamic condition, or attribute condition over r^* .

Definition 42 (*Refinement Primitive*).

A *refinement primitive* is a *DeleteEdge*, *CreateEdge*, *ReplaceNode*, *CreateNode*, *DeleteCorr*, *CreateCorr*, or *AddCondition* refinement.

After defining the general syntactical structure of a refinement and the set of atomic primitives that are to be supported in the refinement language, the following definition introduces the concept of a *network* of multiple and connected refinements.

Definition 43 (*Refinement Network*).

A *Refinement Network* is an acyclic graph $\mathcal{N}(V, E, s, t)$ where each node $n \in V$ in the network is an attributed triple rule and each edge $e \in E$ indicates that $s(e)$ *refines* $t(e)$ in the sense of Def. 34.

An additional set $\mathcal{A} \subset V$ specifies which rules in the network are *abstract*, i.e., are only to be used to factor out commonalities in rules and are not intended to be part of the resulting specification.

Example 25 (*Refinement network for running example*). —

Figure 4.32 depicts a refinement network for our running example. As we have only defined the syntactical structure of refinements, the following discussion of the refinement network can for now only appeal to the reader’s intuition and will be studied in detail in the following sections.

In addition to 8 nodes representing the 8 TGG rules from Ex. 20 and Ex. 21, the network contains two additional nodes *GotoWithPrevRule* and *GotoPaintBasicRule*, which are both marked as abstract (rule names are in italics).

The refinement network is used to factor out commonalities in the rules and will be flattened successively until all refinement relations are “executed” resulting in a set of TGG rules without any refinements (a normal TGG is thus a flattened refinement network, i.e., a refinement network without any edges).

As both rules *PaintGotoRule* and *GotoGotoRule* both share a lot in common (cf. Ex. 20), it is to be expected that they both refine common basis rules. In one case, the factored out commonality *FirstGotoRule* is itself a rule in the resulting TGG, while in the other case, *GotoWithPrevRule* is a rule that is only formed for reuse in *PaintGotoRule* and *GotoGotoRule*. Such *abstract* rules are removed from the flattened network and are not part of the resulting TGG.

The commonality factored out in *GotoPaintBasicRule* is also not surprising as both rules *GotoPaintRule* and *GotoModalPaintRule* handle paint commands.

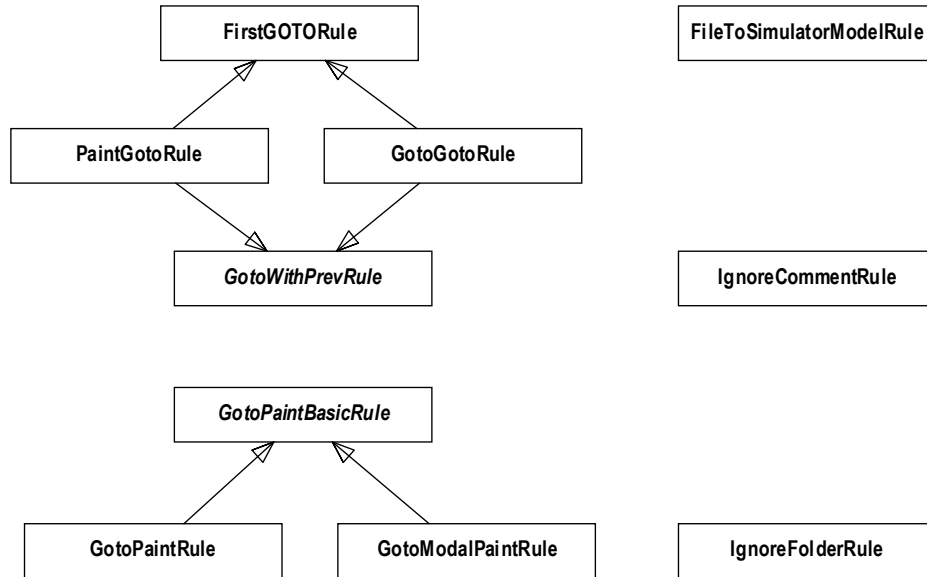


Figure 4.32: Refinement network for running example

The next step is now to define how a general refinement, of the form introduced in Def. 34, can be decomposed into a set of refinement primitives (Def. 42). These primitives represent atomic steps that are then executed in a certain order to actually perform *refinement* and produce a new rule from its basis rules, denoting a step in the process of flattening a refinement network to yield a normal TGG.

4.3.2 Semantics: Decomposing Refinements into Sets of Refinement Primitives

To formalize the semantics of rule refinement, we start by defining how a given refinement can be decomposed into refinement primitives (cf. Def. 42).

We shall do this constructively by providing a coupled grammar that builds up the syntax of a refinement and the corresponding set of refinement primitives simultaneously. The semantics of the given refinement will then be provided by executing the set of refinement primitives in a certain order, resulting in a higher-order transformation that transforms the basis rule to a new refined rule.

To simplify the discussion, we shall only handle the case where all elements in Δ_R and R^* have a pre-image in Δ_L and L^* respectively. This distinction only changes the variant of an induced refinement primitive without affecting its “type”, and is thus irrelevant for the construction process. For instance, an induced DeleteEdge remains a DeleteEdge but can be any of the variants depicted in Fig. 4.25. For presentation purposes, therefore, we focus in the following on $R \leftarrow \Delta R \rightarrow R^*$.

Every refinement $\Delta(r^*, r)$ can be constructed in three steps: (i) source and target components of the refinement are constructed yielding P_S, P_T , (ii) the correspondence components are constructed yielding P_C , (iii) finally, all conditions over r^* are added to P_A as AddCondition primitives.

Step (i) is depicted in Fig. 4.33 for the source components (process is analogous for target components). Note that the start refinement is $R_S \leftarrow \emptyset \rightarrow \emptyset$ as this corresponds to an empty set of refinement primitives. Using rules in a grammar ΔG_S , Δ_{R_S} and R_S^* are built up together with the induced set P_S of source refinement primitives.

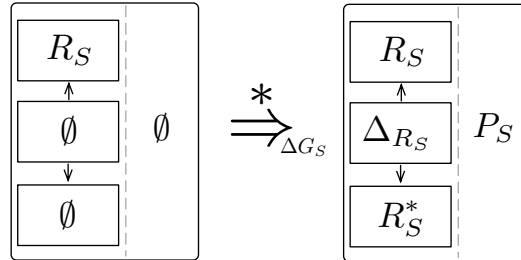


Figure 4.33: A coupled grammar is used to derive source components of refinement and corresponding set of source refinement primitives simultaneously

The five rules in ΔG_S are depicted in Fig. 4.34. Each rule consists of a graph triple rule to the left that builds up the syntactic structure of the refinement, and a rule to the right of the form $L := R$, where L and R are sets of refinement primitives. For conciseness, induced refinement primitives are represented as:

`<type of primitive>("<label of relevant node/edge>")"`

as their exact structure can be inferred from the graph triple rule.

Rules I and II create a node and an edge in R_S^* , respectively. As we assume that labels of elements are unique in rules, this corresponds to adding a CreateNode and CreateEdge source refinement primitive, respectively.

Rules III and IV both add a node to Δ_{R_S} . When creating such a node with label n , two cases are possible: either there exists another node with label m in Δ_{R_S} for which there is an edge with label e between their images in R_S (in this case Rule III is applicable and IV not), or there exists no such node m (in this case Rule IV is applicable and III not). As Rule III requires an existing node with label n in R_S^* , there must already exist a $\text{CreateNode}(n)$ in P_S when Rule III is applicable. In this case, the $\text{CreateNode}(n)$ primitive is deleted and replaced by a $\text{ReplaceNode}(n)$ and a $\text{DeleteEdge}(e)$. Rule IV extends P_S in a similar fashion, only that no DeleteEdge primitive is added. Intuitively, edges are deleted by repeating both source and target nodes of the edge but omitting the edge itself in the refining rule.

Finally, Rule V adds an edge e to Δ_{R_S} . The effect on P_S is twofold: the existing $\text{DeleteEdge}(e)$ must be removed as the edge is now repeated, and the existing $\text{CreateEdge}(e)$ is also deleted as the edge e in R_S^* is now identified with e in Δ_{R_S} .

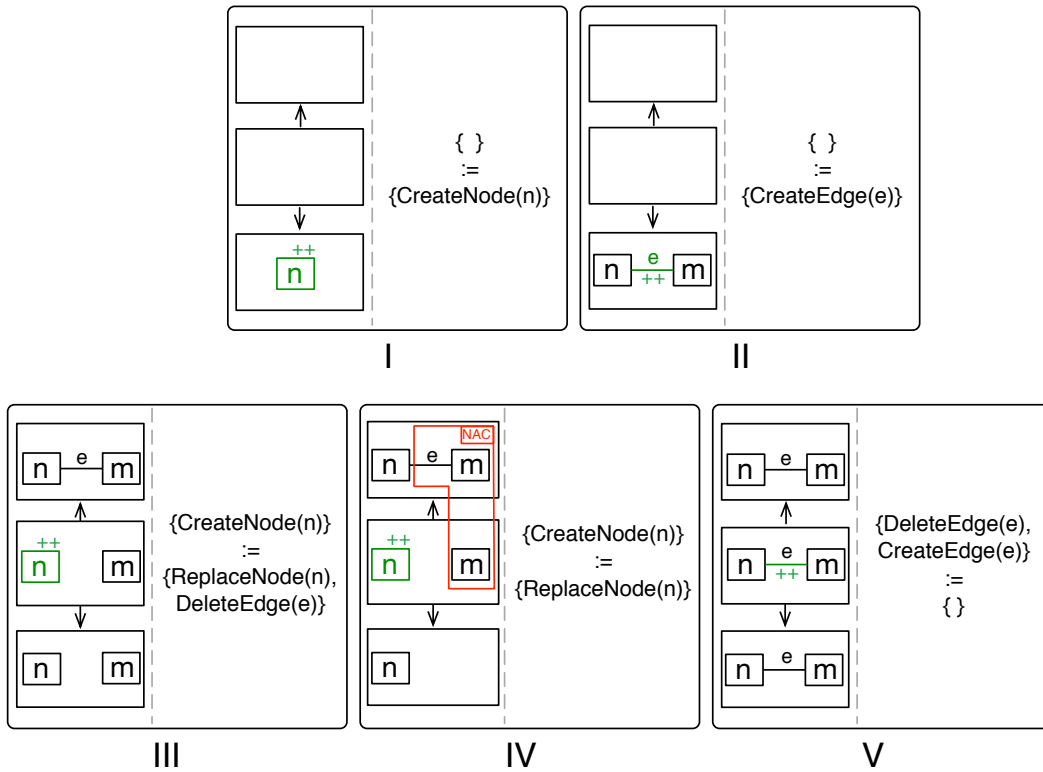


Figure 4.34: Coupled grammar ΔG_S for creating source refinement primitives

Example 26 (*Refinement decomposition in source and target domains*). —

To demonstrate the construction process, Fig. 4.35 depicts the target components of a basis rule r (above the black line) and a refining rule r^* (below the black line) taken from a small excerpt of our running example.

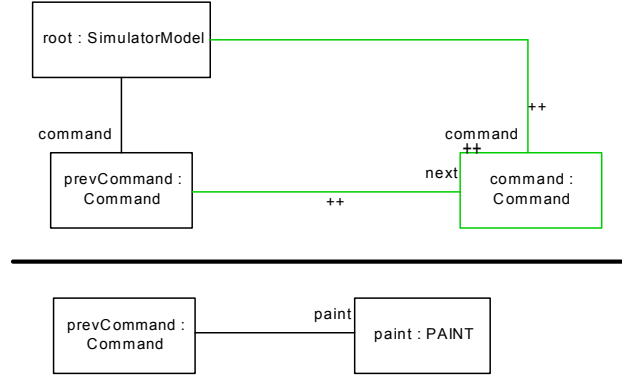


Figure 4.35: Decomposition in target refinement primitives

Using rules in ΔG_S and starting with $R_T \leftarrow \emptyset \rightarrow \emptyset$, the refinement can be constructed as follows (showing P_S in each derivation step):

$$\begin{aligned}
 \{ \} &\xRightarrow{I} \{ \text{CreateNode}(\text{prevCommand}) \} \xRightarrow{I} \\
 &\{ \text{CreateNode}(\text{prevCommand}), \text{CreateNode}(\text{paint}) \} \xRightarrow{II} \\
 &\{ \text{CreateNode}(\text{prevCommand}), \text{CreateNode}(\text{paint}), \text{CreateEdge}(\text{paint}) \} \xRightarrow{IV} \\
 &\{ \text{ReplaceNode}(\text{prevCommand}), \text{CreateNode}(\text{paint}), \text{CreateEdge}(\text{paint}) \}
 \end{aligned}$$

The next step is now to build up the correspondence components for a given refinement and simultaneously derive the set P_C of refinement primitives. This process is depicted in Fig. 4.36 using another coupled grammar ΔG_C . This time around, the start refinement is the state attained after constructing both source and target components of the refinement using ΔG_S and ΔG_T .

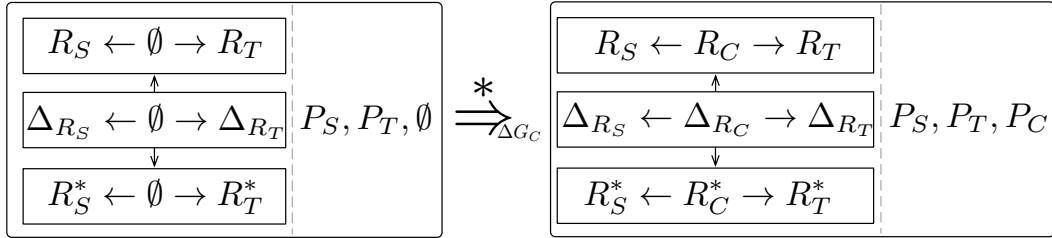


Figure 4.36: Construction of correspondence components and refinement primitives

For presentation purposes, we shall not handle edges in the correspondence components of rules. Rules required for this can be specified analogously to Rules II and V in ΔG_S and do not provide any further insights. Figure 4.37 depicts the coupled grammar ΔG_C consisting of four rules required to perform this step.

Rule I creates a correspondence node in R_C^* and adds a *CreateCorr* refinement to P_C . Conversely, Rule II creates a correspondence node to R_C and a *DeleteCorr* to P_C . Intuitively, therefore, correspondence nodes are handled similar to edges in the source and target components; they can be deleted by repeating source and target nodes but omitting the correspondence “link” in the refining rule.

Rule III handles the case where a correspondence node is to be created in R_C but between source and target nodes that are not *both* identified with nodes in R_C^* via Δ_{R_C} . This condition is enforced with a NAC and means that the set P_C of primitives is not changed at all.

Finally, Rule IV creates a correspondence node in Δ_{R_C} meaning that the correspondence link is to be neither created nor deleted. P_C is thus adjusted appropriately to reflect this.

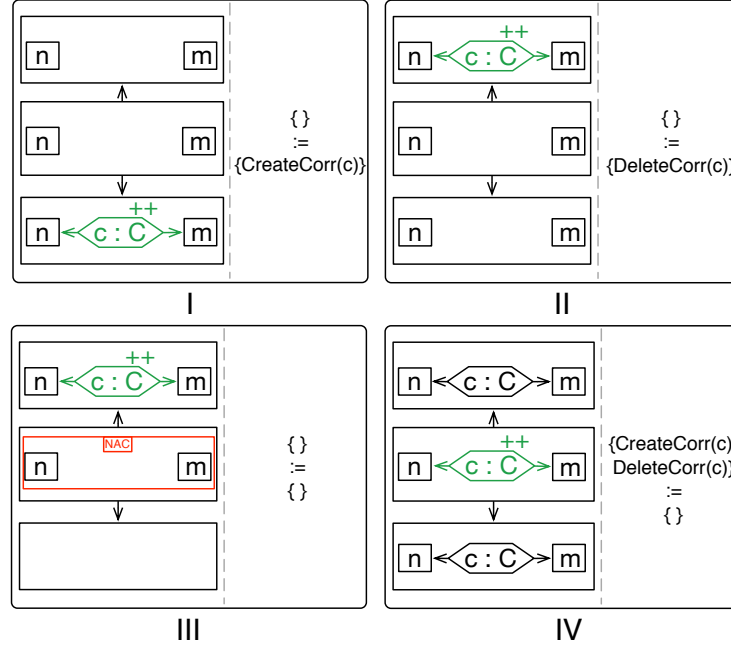


Figure 4.37: Coupled grammar ΔG_C for correspondence components and P_C

Example 27 (*Refinement decomposition for running example*). —————

Figure 4.38 depicts an excerpt of the refinement network for our running example already introduced in Fig. 4.32. The two refining rules *GotoPaintRule* and *GotoModalPaintRule* refine their common basic rule *GotoPaintBasicRule*.

For $\Delta(\text{GotoPaintRule}, \text{GotoPaintBasicRule})$, we get:

$$\begin{aligned}
 P_S &= \{\text{ReplaceNode}(\text{colourNode})\} \\
 P_T &= \{\text{ReplaceNode}(\text{paint})\} \\
 P_C &= \emptyset \\
 P_A &= \{\text{AddCondition}(\text{stringToNumber}(\dots))\}
 \end{aligned}$$

The two replaced nodes do not change anything in the basis rule and are repeated for readability as the attribute condition references their attributes.

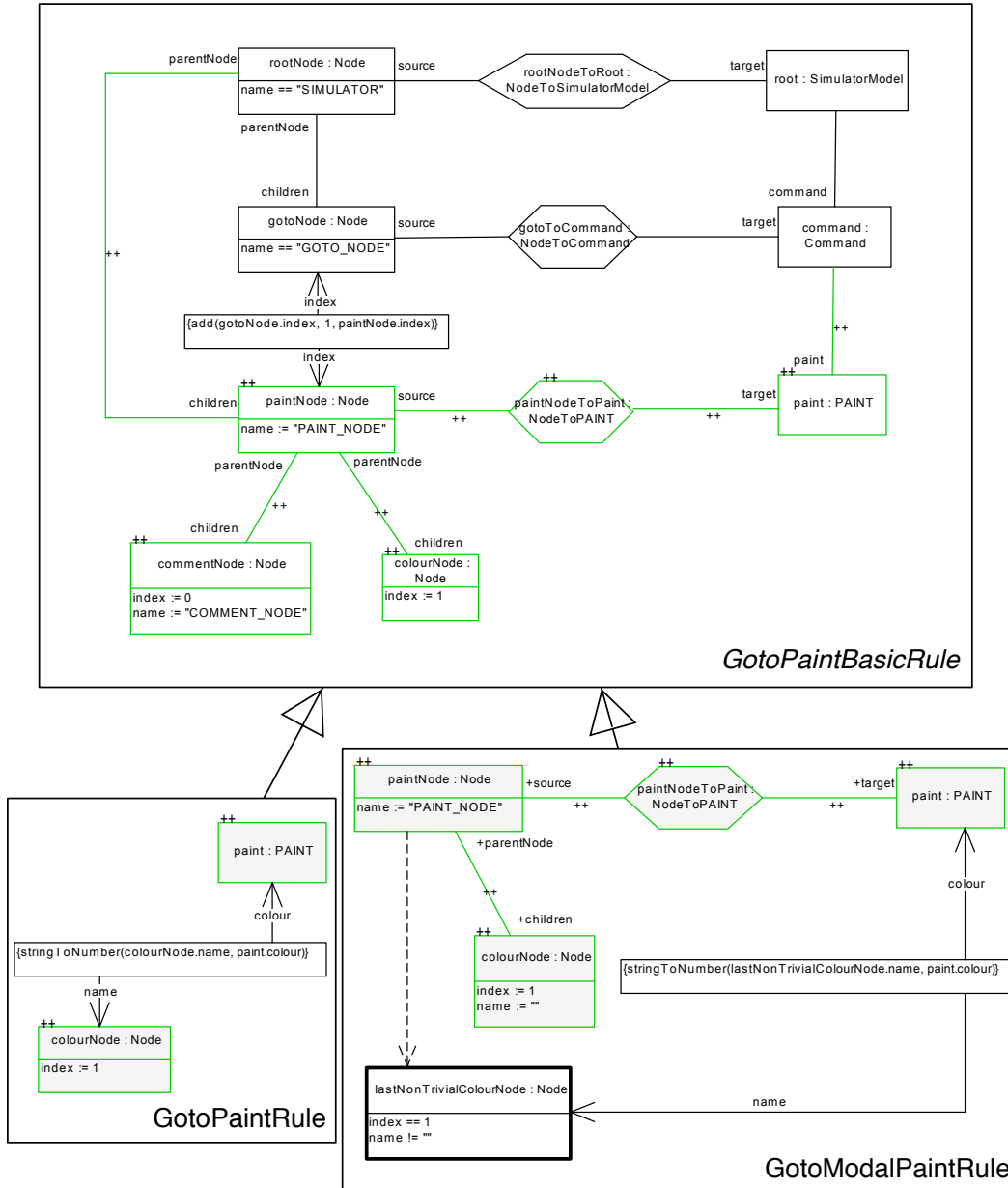


Figure 4.38: Excerpt of refinement network for running example

For $\Delta(\text{GotoModalPaintRule}, \text{GotoPaintBasicRule})$,⁸ we get:

$$\begin{aligned}
 P_S &= \{\text{ReplaceNode}(\text{paintNode}), \text{ReplaceNode}(\text{colourNode}), \\
 &\quad \text{createNode}(\text{lastNonTrivialColourNode})\} \\
 P_T &= \{\text{ReplaceNode}(\text{paint})\} \\
 P_C &= \emptyset \\
 P_A &= \{\text{AddCondition}(\text{stringToNumber}(\dots)), \\
 &\quad \text{AddCondition}(\text{lastNonTrivialColourNode.name} \neq ""), \\
 &\quad \text{AddCondition}((i, \beta))\}
 \end{aligned}$$

In this case, note that $\text{ReplaceNode}(\text{colourNode})$ does indeed change the basis rule, as an additional attribute edge name is added connecting colourNode to the attribute value $""$. This is necessary for creating the trivial colour node for a modal paint command. Also note that the edges between paintNode and colourNode , as well as the correspondence link paintNodeToPaint *must* be repeated in $\text{GotoModalPaintRule}$ to avoid inducing DeleteEdge and DeleteCorr primitives, respectively.

The following derivation in ΔG_C explains why $P_C = \emptyset$ for $\Delta(\text{GotoModalPaintRule}, \text{GotoPaintBasicRule})$:

$$\begin{aligned}
 \{ \} &\xRightarrow{I} \{ \text{CreateCorr}(\text{paintNodeToPaint}) \} \xRightarrow{II} \\
 &\{ \text{CreateCorr}(\text{paintNodeToPaint}), \text{DeleteCorr}(\text{paintNodeToPaint}) \} \xRightarrow{III} \\
 &\{ \text{CreateCorr}(\text{paintNodeToPaint}), \text{DeleteCorr}(\text{paintNodeToPaint}) \} \xRightarrow{III} \\
 &\{ \text{CreateCorr}(\text{paintNodeToPaint}), \text{DeleteCorr}(\text{paintNodeToPaint}) \} \xRightarrow{IV} \\
 \{ \} &= P_C
 \end{aligned}$$

As a final remark on notation, note that all nodes in refining rules, with labels that are also present in a basis rule, are highlighted with a light grey shading.

Given $\Delta(r^*, r)$, Alg. 1 can be used as a strategy to derive the corresponding set of refinement primitives P_S, P_T, P_C, P_A .

Theorem 1 (*Soundness of Refinement Decomposition*).

Given an arbitrary refinement $\Delta(r^*, r)$, decomposition in sets of refinement primitives P_S, P_C, P_T, P_A according to Alg. 1 is possible and unique.

Proof. (Sketch) We have to show two things: (i) *existence*, i.e., that the decomposition in primitives is always possible, and (ii) *uniqueness*, i.e., that the resulting sets of primitives is unique.

To argue existence, we need to show that the canonical derivation derived by Alg. 1 is always possible. The given strategy is able to construct every $\Delta(r^*, r)$ as all nodes and edges in all components are systematically created. As $\Delta(r^*, r)$ consists of finite components, it also terminates showing existence as required.

Although the canonical derivation always exists, it is not unique as multiple rules can be applicable at the same time. One can, for example, decide to create

⁸ Let (i, β) be the dynamic condition indicated by the dashed arrow in $\text{GotoModalPaintRule}$

edges as soon as possible and not after all nodes have been created. It is crucial to show that the resulting set of refinement primitives certainly does not depend on such local decisions. Fortunately, showing that the induced (graph) transformation system for a given (graph) grammar has this property, referred to as *confluence*, is a well-known problem for which there exists a sufficient condition (cf., e.g., Thm. 3.34 in [28]). One must show that $\Delta G_S, \Delta G_T$, and ΔG_C have no *critical pairs*. The interested reader is referred to [28] and Chap. 5 of this thesis, where this technique will be discussed in detail.

Intuitively, whenever two rules in $\Delta G_S, \Delta G_T$, or ΔG_C are both applicable, they do not conflict each other and the order of application can be swapped without changing the result. This is trivial to check for almost all pairs of rules apart from III/IV in $\Delta G_S/\Delta G_T$, and II/III in ΔG_C . In these cases, where derivations with the rules could potentially conflict each other, the NACs in the rules ensure that only one of the derivations is valid. According to [28], therefore, the grammars are all confluent and the resulting sets of refinement primitives is the same for every valid derivation that leads to the same refinement structure. This shows uniqueness as required. \square

Algorithm 1 Refinement Decomposition

Input: A refinement $\Delta(r^*, r)$

Output: Sets of refinement primitives P_S, P_C, P_T, P_A

1. Start with $R_S \leftarrow \emptyset \rightarrow \emptyset$.
 2. Apply Rule I in ΔG_S (Fig. 4.34) to create every node in R_S^* .
 3. Apply Rule II in ΔG_S to create every edge in R_S^* .
 4. Apply Rule III or IV in ΔG_S to create every node in Δ_{R_S} . Note that either III or IV is always applicable for every node, but never both.
 5. Apply Rule V in ΔG_S to create every edge in Δ_{R_S} .
 6. Repeat 1 - 5 for target components analogously.
 7. Apply Rule I in ΔG_C (Fig. 4.37) to create every correspondence node in R_C^* .
 8. Apply Rule II or III in ΔG_C to create every node in R_C . Note that either II or III is always applicable for every node, but never both.
 9. Apply Rule IV in ΔG_C to create every correspondence node in Δ_{R_C} .
 10. For every condition a over r^* , add an $\text{AddCondition}(a)$ primitive to P_A .
-

4.3.3 Semantics: Resolving a Set of Refinement Primitives

After formalizing how refinement primitives are specified, the following algorithm states how each refinement primitive is executed or *resolved*.

Algorithm 2 Refinement Primitive Resolution

Given a triple rule $r : L \rightarrow R$, a refinement primitive $\Delta(r^*, r)$ is *resolved* to yield a new rule $r^*(r)$ from r by executing the corresponding higher-order model transformation given in pseudo code as follows (analogously for target primitives):

DELETEEDGE(E): Remove e from E_{R_S} and, if $e \in E_{L_S^*}$, from E_{L_S} . Adjust source and target functions appropriately by removing entries for e .

CREATEEDGE(E): Add e to E_{R_S} and, if $e \in E_{L_S^*}$, also to E_{L_S} . Adjust source and target functions appropriately by adding entries for e .

REPLACENODE(N): To replace a node m with n , transfer all incident graph and attribute edges from m to n whilst retaining type conformity. If this is not possible, abort (primitive can not be resolved). Remove m from and add n to V_{R_S} (repeat for V_{L_S} if $n \in V_{L_S^*}$).

DELETECORR(C): Remove c from V_{R_C} and, if $c \in V_{L_C^*}$, from V_{L_C} . Adjust graph morphisms between source/target and correspondence components appropriately by removing entries for c .

CREATECORR(C): Add c to V_{R_C} and, if $c \in V_{L_C^*}$, also to V_{L_C} . Adjust graph morphisms between source/target and correspondence components appropriately by adding entries for c .

ADDCONDITION(A): Transfer a from r^* to r . If this is not possible, e.g., required elements have been deleted, abort (primitive can not be resolved).

Algorithm 2 specifies the executable, atomic higher-order transformation each primitive represents. Based on this, we are now able to define the transformation a refinement represents via decomposition in primitives and execution of the primitives in a fixed order.

Note that we assume/demand from the description in Alg. 2 that the order in which primitives of the same kind are executed *in a single step* has no effect on the result and that the groups of primitives are actually executable in the order prescribed by Alg. 3.

To show this formally, one could represent Alg. 2 also as a graph grammar and investigate inter-rule dependencies in detail as we have done for refinement decomposition. We shall, however, draw the line here in this thesis and refer the interested reader to the actual implementation with programmed graph transformations, which is part of the model transformation tool eMoflon.

Algorithm 3 Refinement Resolution

A refinement $\Delta(r^*, r)$ is *resolved* to yield a new rule $r^*(r)$ by decomposing it into sets of primitives P_S, P_C, P_T, P_A according to Alg. 1 and resolving the resulting set of refinement primitives according to Alg. 2 in the following order:

1. All *DeleteCorrs* in P_C
2. All *DeleteEdges* in P_S and P_T
3. All *ReplaceNodes* in P_S and P_T
4. All *CreateNodes* in P_S and P_T
5. All *CreateEdges* in P_S and P_T
6. All *CreateCorrs* in P_C
7. All *AddConditions* in P_A

Example 28 (*Refinement for excerpt from running example*). —

Consider the excerpt of the refinement network from our running example depicted in Fig. 4.38. The corresponding sets of refinement primitives for both refinements were determined in Ex. 27.

Resolving these primitives according to Alg. 2 and in the order given by Alg. 3 produces the original rules *GotoPaintRule* and *GotoModalPaintRule* as introduced in Ex. 20 and 21.

4.3.4 Semantics: Handling Multiple Refinement

Our next goal is to specify how *multiple refinement* is to be handled. The following definition introduces a merge operator defined on rules, which is used to merge all basic rules of a refining rule in a refinement network, before performing the refinement on the merged parent rule according to Alg. 3.

Definition 44 (*Merge Operator \oplus*).

Given a finite set $\{r_1, r_2, \dots, r_n\}$ of rules $r_i : L_i \rightarrow R_i$, and a morphism $\mu_R : R_{1,2,\dots,n} \rightarrow R$, $[r : L \rightarrow R] = \oplus(r_1, r_2, \dots, r_n)$ can be constructed as follows:

$\{L_{1,2,\dots,n}, \rho_{l_1}, \rho_{l_2}, \dots, \rho_{l_n}\}$ and $\{R_{1,2,\dots,n}, \rho_{r_1}, \rho_{r_2}, \dots, \rho_{r_n}\}$ are constructed as the *co-products* (generalized disjoint union cf. Def. A.26 in [28]) of L_1, L_2, \dots, L_n and R_1, R_2, \dots, R_n , respectively.

The typed attributed triple morphism $\mu_R : R_{1,2,\dots,n} \rightarrow R$ must be provided (e.g., via a labelling function specified by the user) and represents the decision which elements are to be regarded as equal and, therefore, merged in R . The morphism $\mu_L : L_{1,2,\dots,n} \rightarrow R$ is obtained via the universal property of the coproduct $\{L_{1,2,\dots,n}, \rho_{L_1}, \rho_{L_2}, \dots, \rho_{L_n}\}$, while L, μ_{L_e} and μ_{L_m} are obtained via an epi-mono factorization of $\mu_L = \mu_{L_m} \circ \mu_{L_e}$. The construction, depicted in Fig. 4.39, is thus uniquely fixed by the (user's) choice of μ_R .

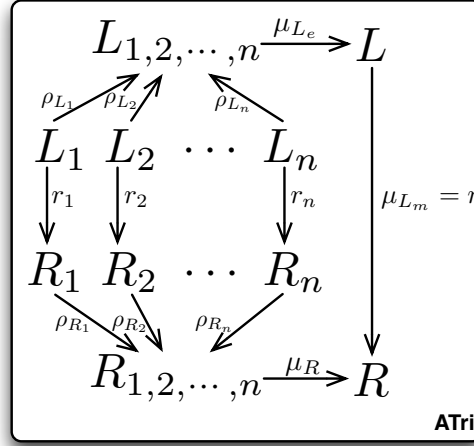


Figure 4.39: Merge Operator

Theorem 2 (*Merge Operator is Sound*).

The merge operator given by Def. 44 is commutative w.r.t. its arguments and uniquely defined for a given μ_R .

Proof. Co-products exist for triple graphs as shown in [6]. The universal property of the co-product construction is used to show the existence of $\mu_L : L_{1,2,\dots,n} \rightarrow R$, which is independent of the order of elements in the set $\{L_1, L_2, \dots, L_n\}$, i.e., the merge operator \oplus is commutative. Every triple morphism can be uniquely decomposed into an epimorphism (surjective part) and a monomorphism (injective part). This property is referred to as *epi-mono factorization* and is shown to hold for **ATri** in [6]. This factorization is used to show the existence and uniqueness (up to isomorphism) of $\mu_L = \mu_{L_e} \circ \mu_{L_m}$. The typed attributed triple morphism μ_{L_e} is an epimorphism (surjective but not necessarily injective) and represents the merging of elements in $L_{1,2,\dots,n}$ induced by μ_R , while μ_{L_m} is a monomorphism (injective but not necessarily surjective) and represents the inclusion of L in R . \square

Example 29 (*Merging two rules*).

To demonstrate how the merge operator is used, Fig. 4.40 depicts the complete merge process for the target components of `FirstGotoRule` (as $r_1 : L_1 \rightarrow R_1$) and `GotoWithPrevRule` (as $r_2 : L_2 \rightarrow R_2$). As the process is defined component-wise, source and correspondence components are merged analogously.

The final rule $r : L \rightarrow R$ is formed from the co-products $L_{1,2}$ and $R_{1,2}$ by merging nodes with the same label, i.e., μ_R is specified via the choice of labels of all elements in the rules. This corresponds exactly to what is done in practice.

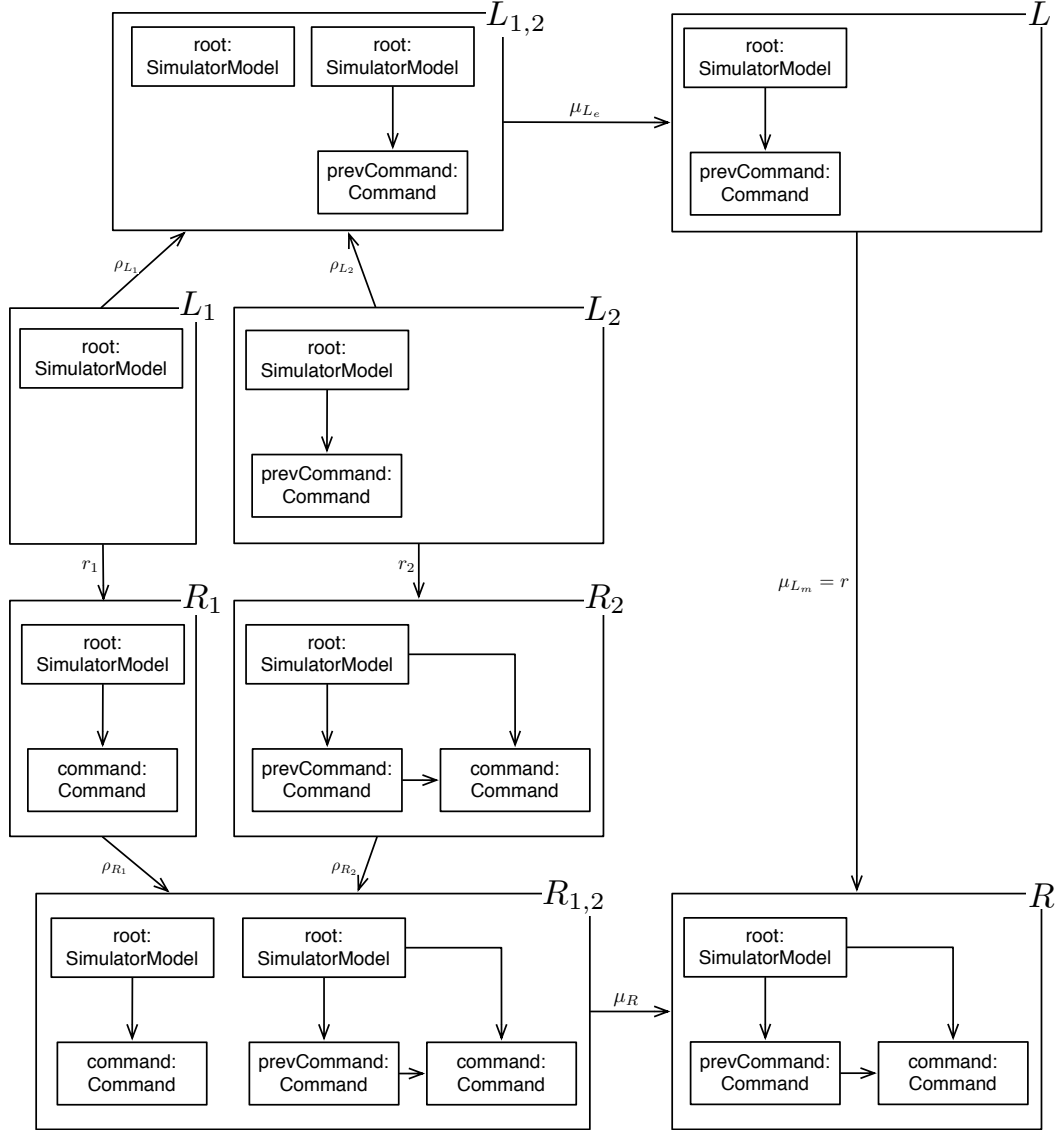


Figure 4.40: Merge operator applied to `FirstGotoRule` and `GotoWithPrevRule`

4.3.5 Semantics: Resolving Refinement Networks

Using the merge operator and refinement resolution, we can now provide an algorithm for resolving a refinement network to a TGG (without refinements):

Algorithm 4 Refinement Network Resolution

Input: A refinement network $\mathcal{N}(V, E, s, t)$

Output: A TGG

A refinement network $\mathcal{N}(V, E, s, t)$ is resolved as follows:

1. Every node r without outgoing edges is regarded as a *resolved* triple rule $r()$.
2. Every node r^* with a single outgoing edge e to a resolved rule $r(\dots)$ is regarded as a refinement: $\Delta(r^* = s(e), r = t(e))$.
3. Every node r^* with multiple outgoing edges e_1, e_2, \dots, e_k to resolved rules $r_1(\dots), r_2(\dots), \dots, r_k()$ respectively, is regarded as a refinement over the result of merging all rules: $\Delta(r^*, r = \oplus(r_1(\dots), r_2(\dots), \dots, r_k(\dots)))$.
4. Every refinement $\Delta(r^*, r(\dots))$ is resolved according to Alg. 3, transforming the refinement network \mathcal{N} to \mathcal{N}' by removing all outgoing edges from r^* , e_1, e_2, \dots, e_k , and replacing r^* with the resolved rule $r^*(r(\dots))$ in the network.
5. As \mathcal{N} is required to be acyclic, there exists a partial order k_0, k_1, \dots, k_l in which the network can be transformed with steps (1) – (4) until there are no edges left, i.e., $\mathcal{N} \xrightarrow{k_1} \mathcal{N}_1 \xrightarrow{k_2} \dots \xrightarrow{k_l} \mathcal{N}_l = (V_{\mathcal{N}_l}, \emptyset)$.
6. A refinement network without any edges is *resolved* and consists only of TGG rules. The final TGG is constructed from a resolved refinement network by excluding all abstract rules.

Theorem 3 (Completeness of Refinement).

A refinement network $\mathcal{N}(V, E, s, t)$ can be resolved to a TGG if (i) all *ReplaceNode* primitives are restricted to using type preserving morphisms, and (ii) if all conditions can be transferred from r^* to r .

If the refinement network can be resolved, the resulting TGG is unique up to isomorphism.

Proof. (Sketch) The refinement network is acyclic so there exists at least one topological order in which the network can be resolved according to Alg. 4 (decomposition is always possible by Thm. 1).

Demanding that all *ReplaceNode* primitives are restricted to using type preserving morphisms and that all conditions can be transferred from r^* to r ensures that all refinement primitives can be resolved.

Although there might be multiple sortings of the network the resolution process for a rule r only depends on its transitive dependencies, which are *before* r in any valid sorting. The merge operator is commutative (Thm. 2), so the resulting TGG is independent of the order in which basis rules are resolved. \square

Example 30 (*Resolving the refinement network for our running example*). —————

We are now ready to resolve the refinement network depicted in Fig. 4.32. All rules without outgoing edges are already resolved: `FileToSimulatorModelRule`, `IgnoreCommentRule`, `IgnoreFolderRule`, and `FirstGotoRule`. We have also resolved the two refining rules `GotoPaintRule` and `GotoModalPaintRule` in Ex. 27.

The remaining excerpt of the network still to be resolved is depicted in detail in Fig. 4.41. Both rules `PaintGotoRule` and `GotoGotoRule` extend the structure in `FirstGotoRule`, merging it with `GotoWithPrevRule` to add a previous command as context. The separation of `FirstGotoRule` from `GotoWithPrevRule` is useful in this case as `FirstGotoRule` is to be a rule in the final TGG, whereas `GotoWithPrevRule` is not. In this simple case, multi-refinement is not absolutely necessary and could be replaced by making `GotoWithPrevRule` refine `FirstGotoRule` instead. The disadvantage of this, however, is that `GotoWithPrevRule` could no longer be used, or *mixed-in*⁹ with a different context that might have nothing to do with `FirstGotoRule`. With multi-refinement, `GotoWithPrevRule` captures adding a previous command as context without enforcing how exactly this is to be used in combination with other rules.

While `GotoGotoRule` only adds an attribute condition to the result of merging both its basis rules, `PaintGotoRule` needs to add another triple together with appropriate attribute conditions for a previous paint command. In both cases, `gotoNode` is replaced, removing the attribute assignment index `:= 0` in the process. This subtle change is important as `FirstGotoRule` is to be applicable only for the first `GOTO` command, while `PaintGotoRule` and `GotoGotoRule` must be applicable for all other `GOTOs` that satisfy the required context.

After resolving the network according to Alg. 4, exactly the same rules as specified in Ex. 20 are produced as the final TGG.

Rule refinement for TGGs is a powerful and flexible tool that must be used with restraint and a focus on readability. In our example, we were able to reduce the number of elements in all rules from 71 to 54 (roughly a 24% reduction). Although this is measurable and certainly indicates a reduction of redundancies in the rules, it says nothing about how intuitive or readable the resulting rule refinement is.

In general, substantial experience is required to produce refinement networks that not only reduce redundancies but even promote readability by factoring out commonalities in a “natural” way, i.e., exactly how a user would expect.

⁹ Term used to refer to composition via multiple inheritance in C++ or Scala

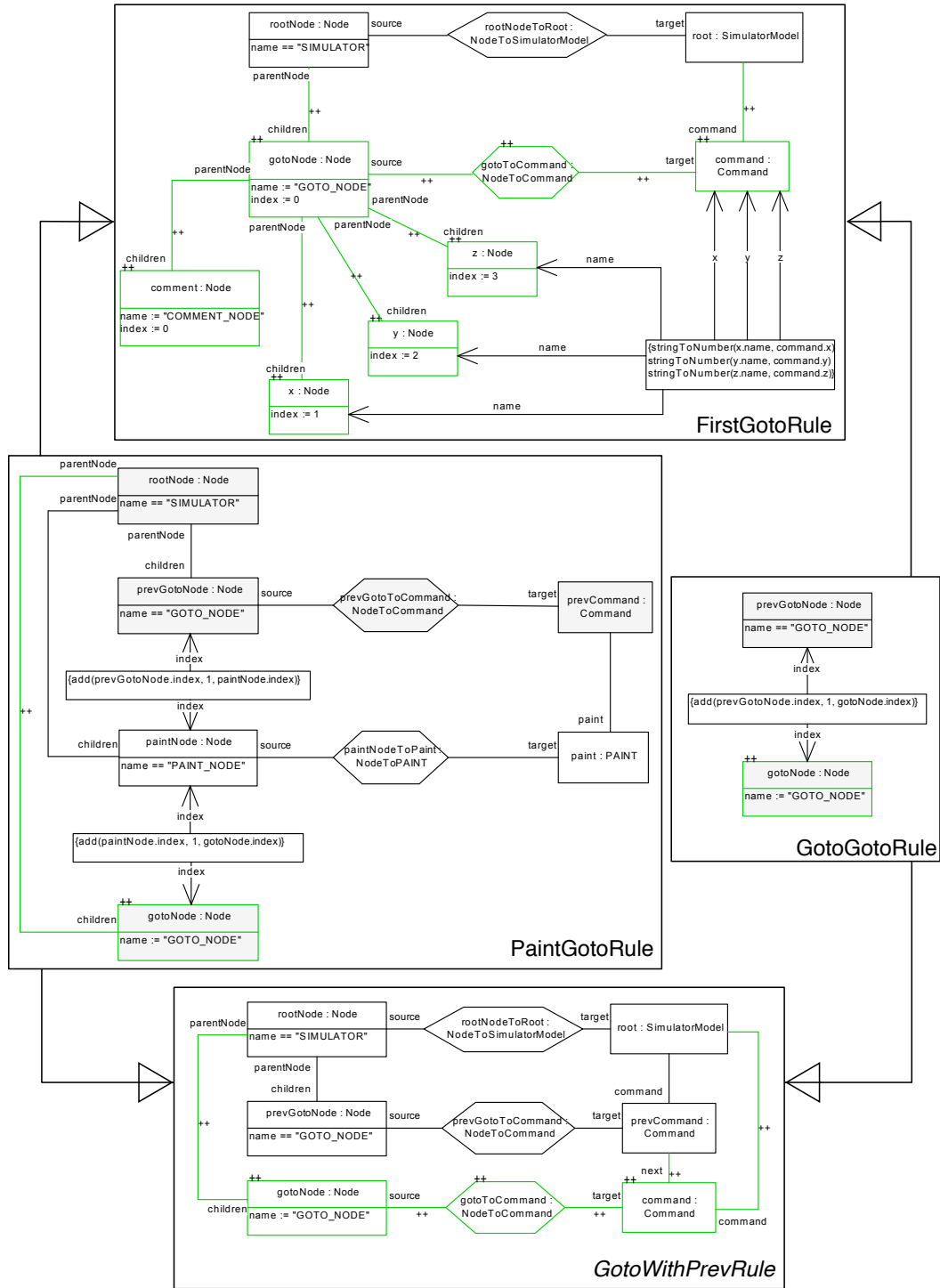


Figure 4.41: Excerpt of refinement network with multi-refinement

4.3.6 Handling Conflicting Refinements when Merging

To conclude this section on rule refinements, the following gives a brief discussion of how interesting corner cases and “conflicts” are handled. In our experience, the behaviour of the merge operator is intuitive and always prefers the conservative “option”, e.g., retain instead of delete, and context instead of create. An evaluation of refinements based on the CME case study is provided in Sect. 7.4.9.

DELETE/CREATE, AND DELETE/RETAIN CONFLICTS FOR EDGES:

A conflict situation arises when two refinements r_1 and r_2 are to be merged, where r_1 deletes a certain edge, but r_2 either creates this edge, or does not delete it. Due to how the merge operator is defined (co-products), adding or retaining the edge wins over deletion in the merged parent.

REPLACE/REPLACE, AND CREATE/CREATE CONFLICTS FOR NODES:

A further conflict situation is given when two refinements r_1 and r_2 both create or replace the same node, but with different types t_1 and t_2 , respectively. Instead of throwing an error immediately, the current implementation first checks if (i) one of the types is a subtype of the other (in this case the subtype wins), (ii) if the types have a common subtype that is then taken, or (iii) if the types have a common supertype that is then used. If none of these cases hold then the process is aborted and an error is reported (conflicting primitives). In all cases, the determined type must of course be compatible with all incident edges in the resulting rule. This is generalized to the case of multiple refinements r_1, \dots, r_n .

CREATE/CONTEXT CONFLICTS:

Given two refinements r_1 and r_2 , a conflict arises when r_1 creates an element (a green node or edge) and r_2 retains this same element as context (a black node or edge). In this case context wins, i.e., the element is taken as context in the result of merging the refinements, again due to how the merge operator is defined.

DERIVATION OF TGG-BASED SYNCHRONIZERS

Up until now, TGGs have been introduced in this thesis as a specification language without discussing how correct and complete incremental synchronizers can be derived from their corresponding specifications.

The required *derivation process* depends on the TGG approach and is typically a compromise between posing minimal restrictions and allowing for a simple, elegant proof of formal properties.

The current *de facto* technique is to *operationalize* TGG specifications by decomposing each TGG rule into a *source* and *forward* rule. These *operational* rules then form the atomic building blocks used by a *control algorithm* to realize forward synchronization. As TGGs are symmetric, backward synchronization with target and backward rules is realized analogously and will not be discussed explicitly in the following.

The goal of operationalization is to decompose a sequence of TGG rule applications into a subsequence of *source rule* applications that only builds up the source model, and a subsequence of *forward rule* applications that creates the correspondence and target models, leaving the source model intact.

As this is essentially what a forward transformation must do given a source model, being able to compose and decompose derivations provides the basis for proving *correctness* (the sequence of source and forward rule applications applied by a forward transformation can be composed to a sequence of TGG rule applications), and *completeness* (every sequence of TGG rule applications can be decomposed to a sequence of source and forward rules) of TGG-based synchronizers.

The derivation process presented in this chapter is based on [29, 32, 67, 71, 88]. Section 5.1 discusses the operationalization of TGG rules, while the control algorithm, together with proofs for the formal properties of TGG-based synchronizers derived via this process, is provided in Sect. 5.2.

The new contribution in this chapter is to incorporate all TGG language extensions from Chap. 4 (attribute conditions, dynamic conditions) into the derivation process and the control algorithm. Together with formal proofs for correctness and completeness under certain, well-defined conditions, this chapter provides the first part of CONTRIBUTION III addressing CHALLENGE III of this thesis as identified in Sect. 1.3:

Provide a constructive, coherent formal underpinning of all concepts and language constructs, which can serve as a guide for a corresponding implementation and static analysis.

5.1 OPERATIONALIZATION OF TGG-RULES

As TGG rules describe the simultaneous evolution of source, correspondence, and target models, they are not particularly suitable for performing *forward* synchronization. Although TGG rules could be directly interpreted and used to this end, formal reasoning about restrictions and properties is greatly simplified by first of all *decomposing* each TGG rule into source and forward rules.

5.1.1 Restrictions for Operationalizability

Application conditions complicate this decomposition and current algorithms, which require an operationalization of TGG rules, restrict the supported class of TGGs to having only NACs and not general (positive) application conditions as introduced in Def. 9.

The class of supported NACs is even further restricted to *source* and *target* NACs, i.e., NACs that are only relevant in either the source or target domain but not in both. Intuitively, such NACs can be trivially *separated* and are thus compatible with a decomposition into source and forward rules. This restriction and the corresponding class of *operationalizable* TGGs is formalized in the following.

Definition 45 (*Source/Target Negative Application Conditions*).

Given a triple rule $r : L \rightarrow R$, a *source* NAC over r is a NAC over r of the form:

$$(n_S, id_{L_C}, id_{L_T}) : L \rightarrow (N_S \xleftarrow{n_S \circ \sigma_L} L_C \xrightarrow{\tau_L} L_T) \text{ as depicted in Fig. 5.1.}$$

Analogously, a *target* NAC over r is of the form:

$$(id_{L_S}, id_{L_C}, n_T) : L \rightarrow (L_S \xleftarrow{\sigma_L} L_C \xrightarrow{n_T \circ \tau_L} N_T).$$

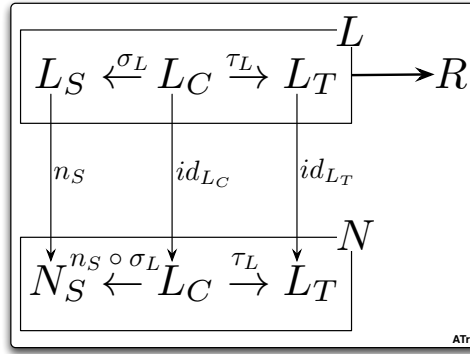


Figure 5.1: Structure of a source NAC

Definition 46 (*Operationalizable TGGs*).

A TGG $= (TG, \mathcal{R})$ is *operationalizable* if $\forall r \in \mathcal{R}$, every application condition for r is either a source or a target NAC.

Example 31 (*Understanding the Restrictions on Application Conditions*). —

To illustrate this restriction on NACs with a concrete example, Fig. 5.2 depicts a TGG with both an allowed source NAC as well as a forbidden NAC.

The TGG is a variant of Ex. 8 with rules (a), (b), (d), (e) unchanged, rule (c') formed by adding a source NAC to the former rule (c), and new rules (f), (g).

Recall that rules (a) and (b) are used to create a chain of corresponding Commands and GOTOs, while rules (d) and (e) ignore y and z arguments.

Rules (c'), (f), and (g) describe how arguments are visualized with corresponding PAINT commands in the simulator model. In contrast to Ex. 8, all coordinates can be visualized with the following two requirements:

1. If x is visualized, it should be visualized before y or z.
2. If y is visualized, it should be visualized before z.

Note that y or z can be ignored using rules (d) or (e), respectively. The NAC used in rule (c') to enforce requirement (1) is a source NAC according to Def. 45 and is therefore valid. The NAC used in rule (f) is, however, neither a source nor target NAC and thus cannot be operationalized.

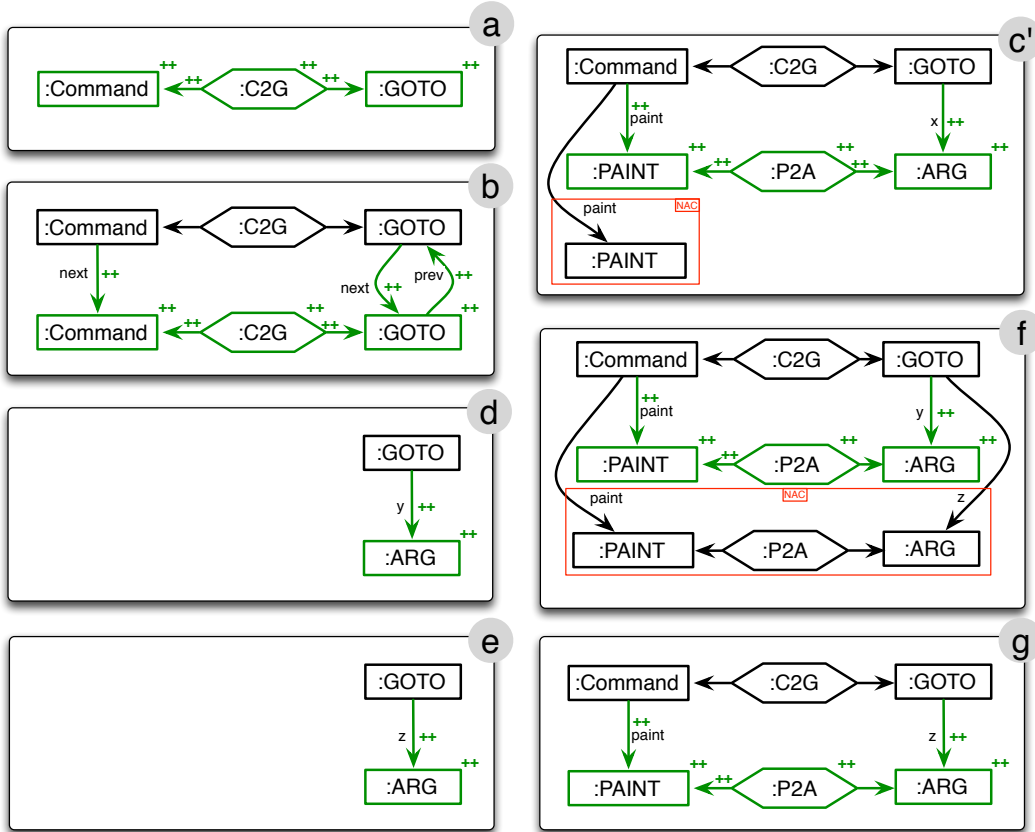


Figure 5.2: Example extended with both allowed and forbidden NACs

To explain why this NAC cannot be naïvely decomposed into a source and target NAC, Fig. 5.3 depicts a target delta to be applied to a triple consisting of a single Command and corresponding GOTO with x and z coordinates. As only the x coordinate has been visualized in the simulator model with a PAINT, i.e., the z coordinate has been ignored, we expect a synchronizer derived from the TGG to be able to propagate the added y coordinate either by ignoring it (rule (d)) or by visualizing it using rule (f). Note that the NAC used in rule (f) does not block rule application as z has been ignored here.

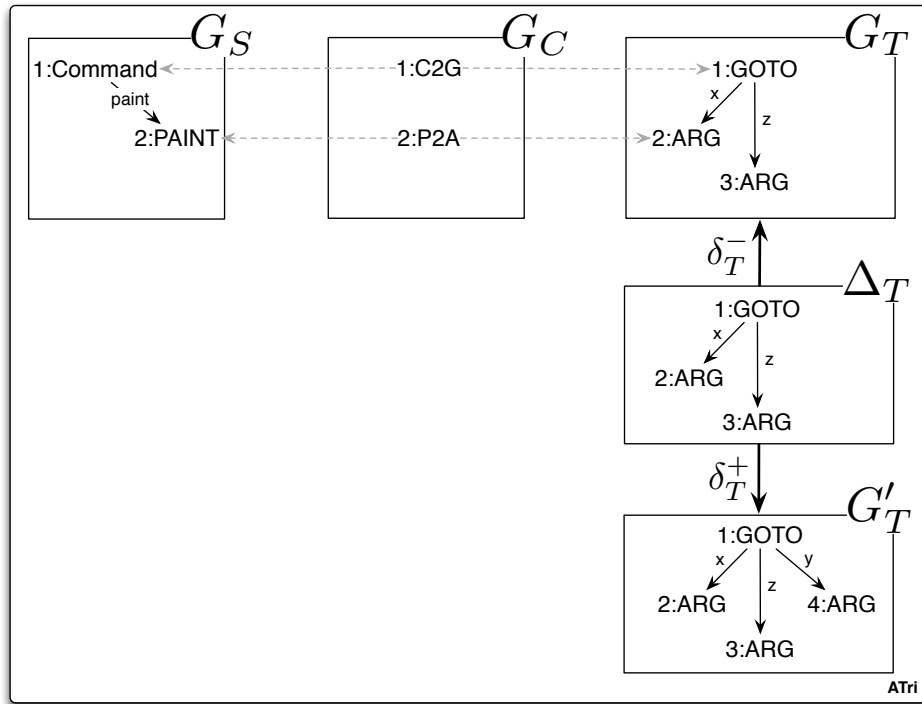


Figure 5.3: Triple and target delta to illustrate expected synchronizer behaviour

Trying to decompose the NAC in rule (f) into source and target NACs does not work; forbidding any other PAINT in the simulation model would mean that y can only be ignored when propagating the target delta. Forbidding a z coordinate in the GOTO model would also mean that y can only be ignored and no longer visualized. As the behaviour of the derived synchronizer differs in both cases from the expected behaviour based on the TGG, this is an example for a NAC that cannot be split into source and target NACs without changing semantics (the language of consistent triples described by the TGG).

To summarize, the restriction to source and target NACs is a real limitation that reduces expressiveness as certain TGGs, such as the set of rules in this example, cannot be supported. As with any grammar-based specification, however, such TGGs can in some cases be reformulated to describe the exact same language using e.g., appropriate context or attribute conditions instead of invalid (neither source nor target) NACs, which prevent operationalization.

5.1.2 Derivation of Source and Forward Rules

The following definition formalizes the decomposition of a single TGG rule into source and forward rules. There are two important points to note:

- Dynamic conditions are restricted to be completely contained in either the source or target domains for the same reasons as NACs, i.e., to simplify the decomposition process.
- To enable a clear and simple formalization, the complete set of attribute conditions of the TGG rule is repeated in both source and forward rules, i.e., is not decomposed in any way. For an efficient implementation, a further analysis of each attribute condition is used to determine conditions that can be safely postponed without violating Def. 47. Such technical details are, however, out-of-scope for this thesis and are abstracted from in the following.

Definition 47 (*Source and Forward Rules*).

Let $TGG = (TG, \mathcal{R})$ be operationalizable and $r : L \rightarrow R \in \mathcal{R}$ be a TGG rule with attributed typed triple graphs L, R over the same term extension $A(X)$ of an algebra A and a set X of variables.

Given $L = L_S \xleftarrow{\sigma_L} L_C \xrightarrow{\tau_L} L_T, R = R_S \xleftarrow{\sigma_R} R_C \xrightarrow{\tau_R} R_T$, and $r = (r_S, r_C, r_T)$, let r be equipped with a set \mathcal{N}_S of source NACs, a set \mathcal{N}_T of target NACs, a set $\mathcal{DC}_S^{\text{pre}}$ of dynamic preconditions over L_S , a set $\mathcal{DC}_S^{\text{post}}$ of dynamic postconditions over R_S , a set $\mathcal{DC}_T^{\text{pre}}$ of dynamic preconditions over L_T , a set $\mathcal{DC}_T^{\text{post}}$ of dynamic postconditions over R_T , and a set $\mathcal{C}_{A(X)}$ of attribute conditions.

Given $SL = L_S \xleftarrow{\emptyset} \emptyset \xrightarrow{\emptyset} \emptyset, SR = R_S \xleftarrow{\emptyset} \emptyset \xrightarrow{\emptyset} \emptyset$, the *source rule* $sr : SL \rightarrow SR$ of r is defined as $sr = (r_S, \emptyset, \emptyset)$ with the sets $\mathcal{N}_S, \mathcal{DC}_S^{\text{pre}}, \mathcal{DC}_S^{\text{post}}, \mathcal{C}_{A(X)}$ of source NACs, dynamic preconditions over L_S , dynamic postconditions over R_S , and attribute conditions, respectively.

Given $FL = R_S \xleftarrow{r_S \circ \sigma_L} L_C \xrightarrow{\tau_L} L_T, FR = R_S \xleftarrow{\sigma_R} R_C \xrightarrow{\tau_R} R_T$, the *forward rule* $fr : FL \rightarrow FR$ of r is defined as $fr = (\text{id}_{R_S}, r_C, r_T)$ with the sets $\mathcal{N}_T, \mathcal{DC}_T^{\text{pre}}, \mathcal{DC}_T^{\text{post}}, \mathcal{C}_{A(X)}$ of target NACs, dynamic preconditions over L_T , dynamic postconditions over R_T , and attribute conditions, respectively.

This *operationalization* process for deriving source and forward rules from a TGG rule is depicted visually in Fig. 5.4. Derivation of *target* and *backward rules* is defined analogously.

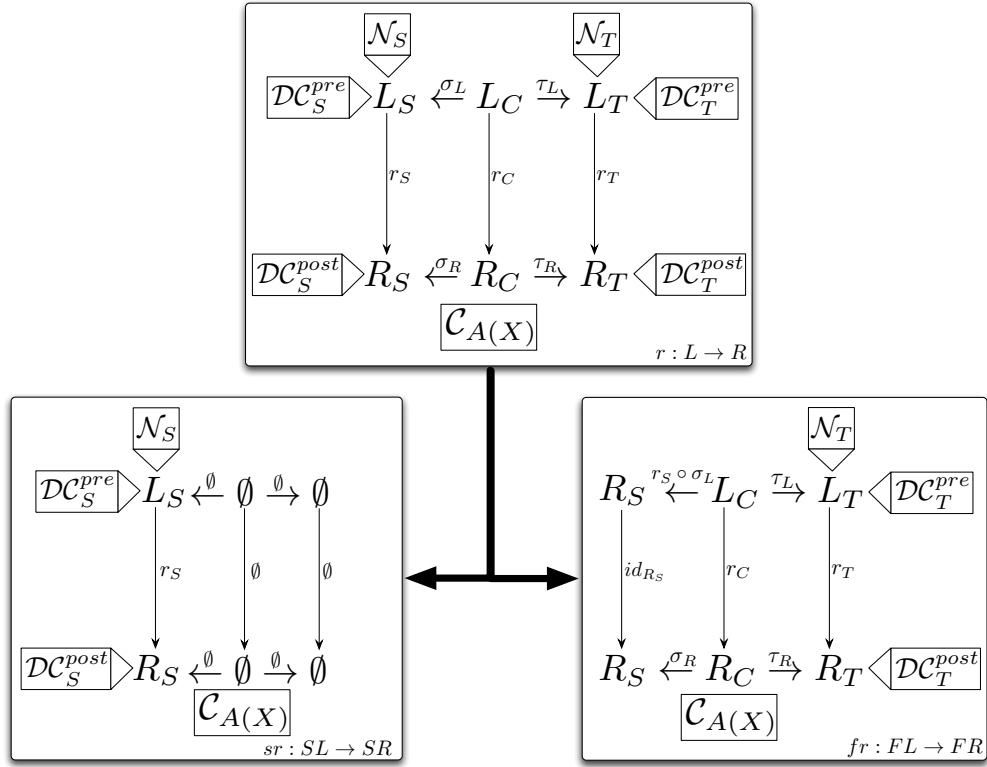


Figure 5.4: Derivation of source and forward rules from a TGG rule

Example 32 (*Derivation of source and forward rules*). —

Figure 5.5 depicts source and forward rules derived from the TGG rule `GotoModalPaintRule` presented previously in Ex. 21. Note how all attribute conditions are repeated in both the source and forward rule in accordance with Def. 47. Of interest is also that the source dynamic post-condition is not present in the forward rule and that all assignments in the source component are converted to assertions. NACs are handled analogously to dynamic conditions.

Intuitively, all elements created by the source rule are demanded as context in the forward rule, and are expected to satisfy all (attribute, negative, and dynamic) application conditions. A bit surprising is probably that the *complete* set of attribute conditions must already be solved in the source rule. This is necessary in general as attribute constraints can range over variables in both the source and the forward rule. The values of all variables in the forward rule are, therefore, already fixed by the source rule application and *must* be reused when applying the forward rule.¹ This induces a notion of compatibility of source and forward rule applications, which must hold to be able to compose them back to a TGG rule application.

¹ For efficiency reasons, this condition is relaxed in practice and attribute constraints are solved as *lazily* as possible. This requires, however, a domain dependency analysis of the constraints, which is out-of-scope for the formalization in this thesis.



5.1.3 Composition and Decomposition of TGG Rules

Our next goal is to formalize the exact condition under which source and target rule applications can be composed to a single TGG rule application. Existing theory concerning the concurrent application of rules over a common match can be applied here. For presentation purposes, NACs are not handled explicitly in the following and the reader is referred to [32] for full details.

The following definition, simplified and formulated for TGG rules, is taken from [28], where it is shown to be a special case of Def. 5.20 and 5.21 in [28].

Definition 48 (*E-Dependency Relation and E-Concurrent Rule in Tri*).

Given TGG rules $r_1 : L_1 \rightarrow R_1$ and $r_2 : L_2 \rightarrow R_2$, a triple morphism $e : R_1 \rightarrow L_2$ is an *E-dependency relation* over r_1 and r_2 if, as depicted in Fig. 5.6, the pushout-complement $(e^* : L_1 \rightarrow L, r_1^* : L \rightarrow L_2)$ exists.

The corresponding *E-concurrent rule* $r_1 *_E r_2$ is defined as $r_2 \circ r_1^* : L \rightarrow R_2$.

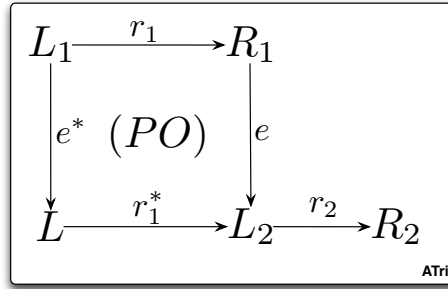


Figure 5.6: E-dependency relation and E-concurrent rule

Intuitively, the existence of L guarantees that it is possible to find a starting point from which both rules can be applied *directly* after each other without anything else happening in-between rule applications. If L , however, does not exist, then r_2 requires context that can only be created after r_1 is applied, but before r_2 is applied. In this case, both rules cannot be merged together to a single rule.

The idea is now to view TGG rules as concurrent rules formed by merging source and forward rules. The following lemma shows that this is indeed possible as a forward rule operates directly on the result of its corresponding source rule without demanding any other context elements that would have to be created in-between rule applications. The starting point for the application of a TGG rule is thus, as can be expected, the state of the complete triple graph before application of the source rule.

Lemma 1 (*TGG Rules are E-Concurrent Rules*).

A TGG rule $r : L \rightarrow FR$ is an E-concurrent rule $sr *_E fr$ of its source rule $sr : SL \rightarrow SR$ and its forward rule $fr : FL \rightarrow FR$.

Proof. An E-dependency relation can be defined over sr and fr as depicted in Fig. 5.7. It follows directly from Def. 48 that $sr *_{\text{E}} fr = r : L \rightarrow FR$ with $\mathcal{C}_{\Lambda(X)}$, all source/target NACs, and all pre/post source/target dynamic conditions according to Def. 47. \square

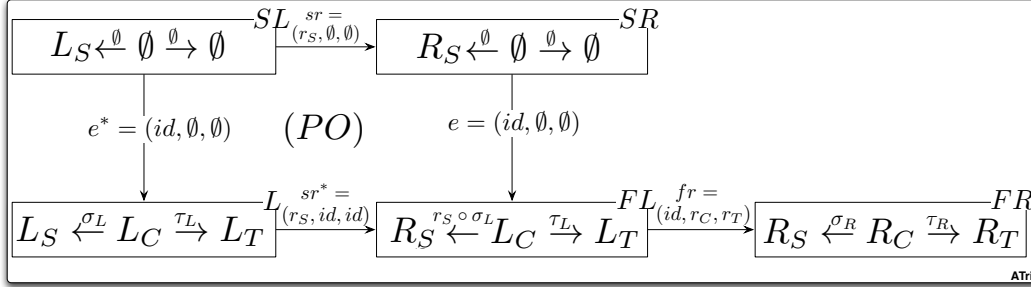


Figure 5.7: A TGG rule is an E-concurrent rule of its source and forward rules

Even if two rules can be merged together to a concurrent rule, it does not mean that *every* pair of derivations with the two rules can be combined to a single derivation with the concurrent rule. This is only the case if the derivations are suitably *related*, as formalized by the following definition.

Definition 49 (*E-Related Derivations*).

Two derivations $G \xrightarrow{r_1 @ m_1} H \xrightarrow{r_2 @ m_2} G'$, as depicted in Fig. 5.8, are *E-related* if $\exists m^* : L \rightarrow G, m_1 = m^* \circ e^*$ and $m'_1 = m_2 \circ e$.

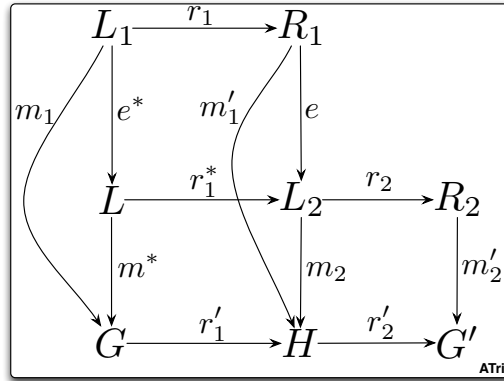


Figure 5.8: Condition for E-related derivations

Intuitively, the matches of E-related derivations must overlap suitably to be able to construct a match for the concurrent rule. Transferring this condition to TGG rules results in the concept of *match consistency* of source and forward rule derivations. This condition, originally formulated by [88] and consequently formalized in a categorical framework by [29], characterizes the condition under which source

and forward derivations can be composed to a derivation of their concurrent TGG rule. The following definition extends the existing notion of match consistency appropriately to cover the new TGG features introduced in this thesis.

To improve readability, graphs in derivations consisting of source and forward rule applications have a double index G_{ij} , where i and j indicate how many source rules and forward rules have been applied up to this point in the derivation, respectively.

Definition 50 (*Match Consistency*).

A derivation $G_{00} \xRightarrow{*} G_{(i-1)j} \xRightarrow{sr_i @ sm_i} G_{ij} \xRightarrow{*} G_{k(i-1)} \xRightarrow{fr_i @ fm_i} G_{ki} \xRightarrow{*} G_{nn}$ of source and forward rules sr_i and fr_i , is *match consistent* if the following holds:

$$\forall 1 \leq i \leq n, 1 \leq j \leq n, i \leq k \leq n, G_{(i-1)j} \xRightarrow{sr_i @ sm_i} G_{ij}, G_{k(i-1)} \xRightarrow{fr_i @ fm_i} G_{ki}$$

1. $(fm_i)_S = d_S \circ m'_S$ as depicted in Fig. 5.9, where $d : G_{ij} \rightarrow G_{k(i-1)}$.
2. The same assignment function $\xi : X \rightarrow A$ is used to determine the data parts of both the source match sm_i , and the forward match fm_i .

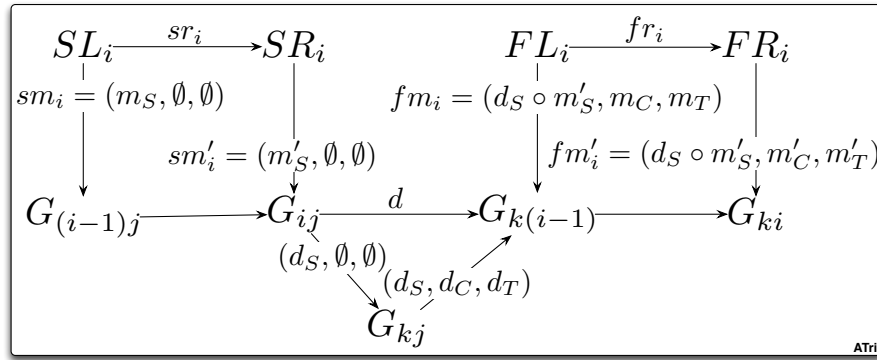


Figure 5.9: Condition for match consistency

Match consistency ensures that the source component of every forward derivation is already fixed by the previous corresponding source derivation. This is demanded by Condition (1) in Def. 50, using the morphism d to embed the state of the graph used for the source derivation into the later state used for the forward derivation. Condition (2) ensures that the chosen values of attributes in the source and forward derivation are compatible and can be combined to fulfil the attribute conditions of the merged TGG rule derivation. Recall from Def. 47 that the exact same set of attribute conditions is used in both source and forward rules, meaning that the forward rule can simply use exactly the same assignment function already completely computed for the source rule derivation. As NACs and dynamic conditions are either over the source or target component but not over both, it is fairly clear that a forward derivation that fulfils Condition (1) also fulfils all source NACs and source dynamic conditions. For the case of NACs, this

is covered in detail by [32]. The same straightforward arguments can be used to show analogous results for dynamic conditions.

The reason for introducing the notion of match consistency is to formalize exactly under what conditions source and forward derivations can be composed. The following lemma shows that match consistency is indeed exactly this condition, i.e., that match consistent pairs of source and forward derivations are E-related according to Def. 49, with respect to their concurrent TGG rule (Def. 48).

Lemma 2 (*Match Consistent Derivations are E-Related for $i = 1$ and $j = 0$*).

A match consistent derivation $G_{00} \xrightarrow{sr@sm_1} G_{10} \xrightarrow{fr@fm_1} G_{11}$ for $i = 1, j = 0$, i.e., for a single source rule $sr = (r_S, \emptyset, \emptyset) : (L_S \xleftarrow{\emptyset} \emptyset \xrightarrow{\emptyset} \emptyset) \rightarrow (R_S \xleftarrow{\emptyset} \emptyset \xrightarrow{\emptyset} \emptyset)$ and forward rule $fr = (id, r_C, r_T) : (R_S \leftarrow L_C \rightarrow L_T) \rightarrow (R_S \leftarrow R_C \rightarrow R_T)$, is E-related according to Def. 49.

Proof. From Lemma 1, the TGG rule depicted in Fig. 5.10

$$r = sr *_E fr = (r_S, r_C, r_T) : (L_S \leftarrow L_C \rightarrow L_T) \rightarrow (R_S \leftarrow R_C \rightarrow R_T)$$

is an E-concurrent rule. To show that $G_{00} \xrightarrow{sr@sm_1} G_{10} \xrightarrow{fr@fm_1} G_{11}$ is E-related, therefore, we must construct a morphism:

$$m^* : (L_S \leftarrow L_C \rightarrow L_T) \rightarrow (G_S \leftarrow G_C \rightarrow G_T)$$

and show that (1) and (2) in Fig. 5.10 commute. $G_{00} \xrightarrow{sr@sm_1} G_{10} \xrightarrow{fr@fm_1} G_{11}$ is match consistent, so $(fm_1)_S = (sm'_1)_S$ follows from Def. 50.

With $fm_1 = (m'_S, m_C, m_T)$, $sm'_1 = (m'_S, \emptyset, \emptyset)$, (2) commutes with $sm'_1 = fm_1 \circ e$. As depicted in Fig. 5.10, $m^* = (m_S, m_C, m_T)$ can be shown to be a triple morphism using standard arguments, i.e., because $sm_1 = (m_S, \emptyset, \emptyset)$ and $fm_1 = (m'_S, m_C, m_T)$ are triple morphisms, and $r'_S : G_S \rightarrow G'_S \in \mathcal{M}$.

Furthermore, match consistency ensures that $m^* : L \rightarrow G$ is a valid match morphism for applying $r_1 : L \rightarrow R$ as the assignment $\xi : X \rightarrow A$, used for both sm_1 and fm_1 , can be reused for m^* to fulfil the set of attribute conditions $\mathcal{C}_{A(X)}$.

Finally, satisfaction of all source and target NACs/dynamic conditions is inherited component-wise by m^* from sm_1 and fm_1 , respectively (cf. [32]).

The triple morphism (m_S, m_C, m_T) is thus a valid choice for m^* and also means that (1) commutes due to $sm_1 = m^* \circ e$. \square

To complete the current analysis of decomposition and composition of TGG rule derivations from source and forward rule derivations, we can now apply the following *Concurrency Theorem*, a central result from [28], simplified and reformulated here for the case of TGG rules.

Based on this theorem, we conclude that TGG rule derivations can always be decomposed into match consistent source and forward rule derivations, and that *only* match consistent source and forward rule derivations can be composed to yield the corresponding TGG rule derivation.

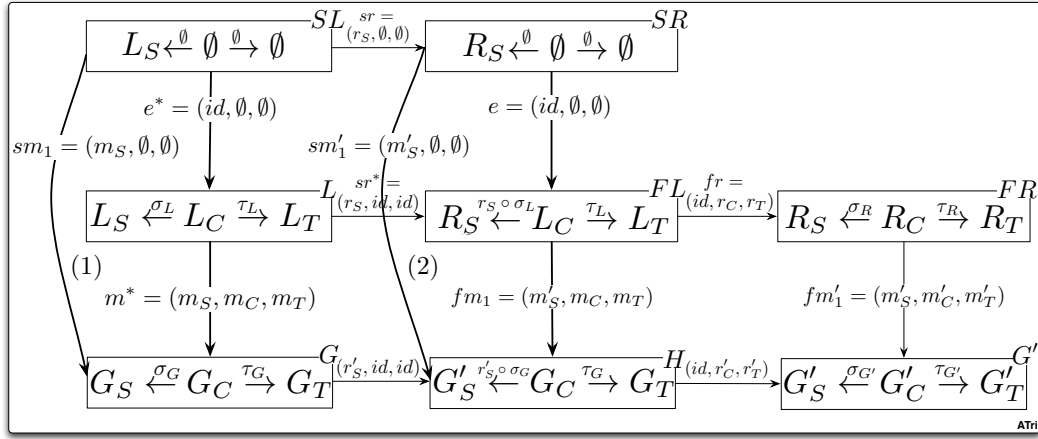


Figure 5.10: A pair of match consistent derivations are E-related

Theorem 4 (Concurrency Theorem).

Let $e : R_1 \rightarrow L_2$ be an E-dependency relation for triple rules $r_1 : L_1 \rightarrow R_1$ and $r_2 : L_2 \rightarrow R_2$, and $r_1 *_{\mathbb{E}} r_2$ the corresponding E-concurrent triple rule.

SYNTHESIS: Given an E-related derivation $G \xRightarrow{r_1} H \xRightarrow{r_2} G'$, there is a *synthesis construction* leading to a direct derivation $G \xRightarrow{r_1 *_{\mathbb{E}} r_2} G'$.

ANALYSIS: Given a direct derivation $G \xRightarrow{r_1 *_{\mathbb{E}} r_2} G'$, there is an *analysis construction* leading to an E-related derivation $G \xRightarrow{r_1} H \xRightarrow{r_2} G'$.

BIJECTIVE CORRESPONDENCE: The synthesis and analysis constructions are inverse to each other up to isomorphism.

Proof. For a detailed proof in a general setting, the interested reader is referred to Theorem 5.23 in [28]. \square

5.1.4 Reordering Derivations to Yield Separated Source and Forward Derivations

Although we now know how to decompose and compose *direct* TGG rule derivations, we do not yet know how to handle derivations of arbitrary length. The basic idea of how to do this, again taken from [88] and formalized in a categorical setting by [29], is to appropriately shuffle a derivation of source rules and forward rules, until all forward rule derivations appear immediately after their corresponding source rule derivations so that Thm. 4 can be applied.

The following definition introduces the concept of *sequential independence*, which will be used to formalize a notion of *swap equivalence* over derivations of source and forward rules.

Definition 51 (*Sequential Independence*).

Two direct derivations $G \xrightarrow{r_1 @ m_1} H \xrightarrow{r_2 @ m_2} G'$ with triple rules $r_1 : L_1 \rightarrow R_1$, $r_2 : L_2 \rightarrow R_2$ are sequentially independent if a morphism $m_2^* : L_2 \rightarrow G$ exists such that $m_2 = r_1' \circ m_2^*$. This is depicted visually in Fig. 5.11.

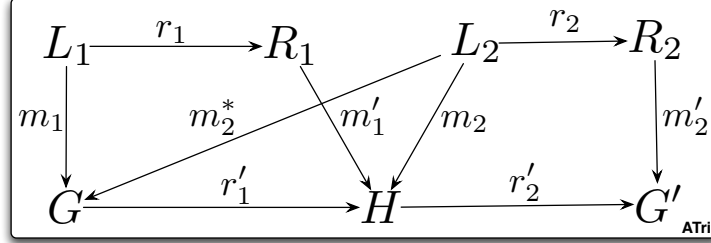


Figure 5.11: Condition for sequential independence

Intuitively, two direct derivations are sequentially independent if a match for the second derivation can already be found in the graph *before* the first derivation is applied. This means that the second derivation does not require anything created by the first and is in this sense independent of the first derivation.

As the goal is to be able to reorder such pairs of independent derivations, the following lemma characterizes which pairs of direct derivations are sequentially independent in a match consistent derivation of source and forward rule.

Lemma 3 (*Sequential Independence of Match Consistent Derivations*).

Let $G_{00} \xRightarrow{*} G_{nn}$ be a match consistent derivation of source and forward rules. For $\forall 1 \leq i \leq n, i < k \leq n$

1. $G_{(k-1)(i-1)} \xrightarrow{sr_k @ sm_k} G_{k(i-1)} \xrightarrow{fr_i @ fm_i} G_{ki}$ is sequentially independent
2. $G_{k(i-1)} \xrightarrow{fr_i @ fm_i} G_{ki} \xrightarrow{sr_{k+1} @ sm_{k+1}} G_{(k+1)i}$ is sequentially independent

Proof. The situation for (1) is depicted in Fig. 5.12. As the derivation is match consistent and $k > i$, the forward match morphism fm_i is of the form:

$fm_i = (d_S \circ m'_{i_S}, m_{i_C}, m_{i_T}) : FL_i \rightarrow G_{k(i-1)}$ for a source co-match morphism

$sm'_i = (m'_{i_S}, \emptyset, \emptyset) : SR_{i_S} \xleftarrow{\emptyset} \emptyset \xrightarrow{\emptyset} \emptyset \rightarrow G_{ij}$, $j < i$, and $d : G_{ij} \rightarrow G_{k(i-1)}$, already applied with $G_{(i-1)j} \xrightarrow{sr_i @ sm_i} G_{ij}$ earlier in the derivation. We can thus conclude that $(d'_S \circ m'_{i_S}, m_{i_C}, m_{i_T}) : FL_i \rightarrow G_{(k-1)(i-1)}$ holds with $d' : G_{ij} \rightarrow G_{(k-1)(i-1)}$, as the source rule sr_k adds additional elements (not required by m'_{i_S}) solely to the source component G_S of $G_{(k-1)(i-1)}$.

The situation for (2) is depicted in Fig. 5.13. As $(m_{(k+1)_S}, \emptyset, \emptyset) : SL_{k+1} \rightarrow G_{ki}$ and the forward rule fr_i does not change the source component G_S , we can conclude that $(m_{(k+1)_S}, \emptyset, \emptyset) : SL_{k+1} \rightarrow G_{k(i-1)}$.

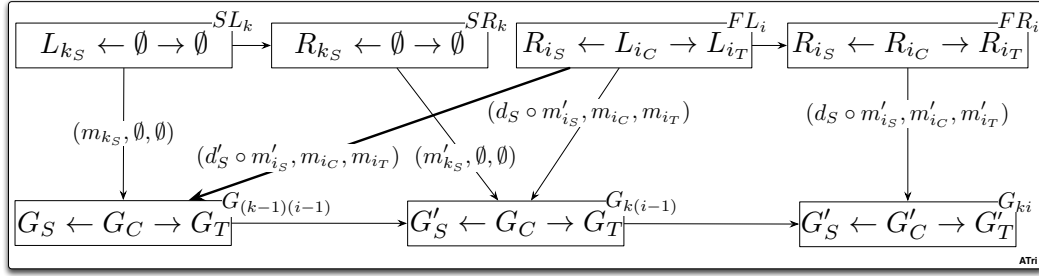


Figure 5.12: Sequential independence of $G_{(k-1)(i-1)} \xrightarrow{sr_k @ sm_k} G_{k(i-1)} \xrightarrow{fr_i @ fm_i} G_{ki}$

This constructs in both cases the morphisms required for showing sequential independence of (1) and (2). \square

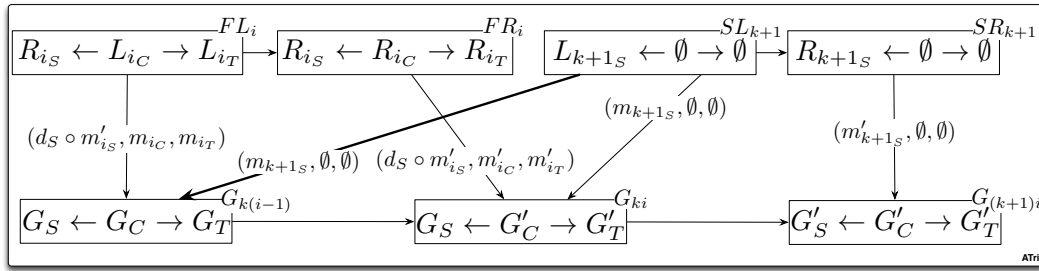


Figure 5.13: Sequential independence of $G_{k(i-1)} \xrightarrow{fr_i @ fm_i} G_{ki} \xrightarrow{sr_{k+1} @ sm_{k+1}} G_{(k+1)i}$

We have identified pairs of sequentially independent derivations because such derivations can be swapped without changing the final resulting graph. The resulting derivation after such a swap is in this sense equivalent to the derivation before swapping. This is guaranteed by the following theorem, also a central result in [28], simplified and formulated here for the case of TGG rules.

Theorem 5 (*Local Church-Rosser Theorem*).

Given two sequentially independent direct derivations $G_1 \xrightarrow{r_1 @ m_1} G_2 \xrightarrow{r_2 @ m_2} G_3$, there are a triple graph G'_2 and two sequentially independent direct derivations $G_1 \xrightarrow{r_2 @ m'_2} G'_2 \xrightarrow{r_1 @ m'_1} G_3$.

Proof. For a detailed proof, the reader is referred to Theorem 5.12 in [28]. \square

We are now ready to state the final result of this section, taken from [29] and extended appropriately, a means of decomposing derivations of TGG rules to match consistent derivations of source and target rules and vice-versa. This is an important result as it means that (i) a synchronizer is correct if it produces a match consistent derivation of source and forward rules, and (ii) it is theoretically possible for synchronizers to be complete as *every* derivation of TGG rules can be decomposed as required.

Theorem 6 (*Composition and Decomposition Theorem*).

Let $TGG = (TG, \mathcal{R})$ be an operationalizable triple graph grammar.

For every rule $r_i \in \mathcal{R}, 1 \leq i \leq n$, let sr_i and fr_i denote the source and forward rule for r_i , respectively.

DECOMPOSITION: For every derivation $G_{00} \xRightarrow{r_1} G_{11} \xRightarrow{r_2} \dots \xRightarrow{r_n} G_{nn}$ of TGG rules, there is a corresponding match consistent derivation $G_{00} \xRightarrow{sr_1} G_{10} \xRightarrow{sr_2} \dots \xRightarrow{sr_n} G_{n0} \xRightarrow{fr_1} G_{n1} \xRightarrow{fr_2} \dots \xRightarrow{fr_n} G_{nn}$ of source and forward rules.

COMPOSITION: For every match consistent derivation $G_{00} \xRightarrow{sr_1} G_{10} \xRightarrow{sr_2} \dots \xRightarrow{sr_n} G_{n0} \xRightarrow{fr_1} G_{n1} \xRightarrow{fr_2} \dots \xRightarrow{fr_n} G_{nn}$ of source and forward rules, there is a corresponding derivation $G_{00} \xRightarrow{r_1} G_{11} \xRightarrow{r_2} \dots \xRightarrow{r_n} G_{nn}$ of TGG rules.

BIJECTIVE CORRESPONDENCE:

Composition and decomposition are inverse to each other.

Proof.

DECOMPOSITION: Given Derivation ① $G_{00} \xRightarrow{r_1} G_{11} \xRightarrow{r_2} \dots \xRightarrow{r_n} G_{nn}$ of TGG rules as depicted in Fig. 5.14 $\xRightarrow{\text{Lemma 1}}$

every TGG rule is an E-concurrent rule of its source and forward rules so each direct derivation $G_{(i-1)(i-1)} \xRightarrow{r_i} G_{ii}$ in (1) can be replaced by the direct derivation $G_{(i-1)(i-1)} \xRightarrow{sr_i * fr_i} G_{ii} \xRightarrow{\text{Thm. 4}}$

there is an analysis construction leading to $G_{(i-1)(i-1)} \xRightarrow{sr_i} G_{i(i-1)} \xRightarrow{fr_i} G_{ii}$, an E-related derivation $\xRightarrow{\text{Lemma 2}}$ this E-related derivation is match consistent.

Repeating this analysis construction for every direct derivation in (1) leads to the match consistent Derivation (2) depicted in Fig. 5.14 $\xRightarrow{\text{Lemma 3}}$

derivations $G_{k(i-1)} \xRightarrow{fr_i @ fm_i} G_{ki} \xRightarrow{sr_{k+1} @ sm_{k+1}} G_{(k+1)i}$ with $k > i$ are sequentially independent $\xRightarrow{\text{Thm. 5}}$ and can be replaced with $G_{k(i-1)} \xRightarrow{sr_{k+1} @ sm_{k+1}} G_{(k+1)(i-1)} \xRightarrow{fr_i @ fm_i} G_{(k+1)i}$ preserving match consistency and sequential independence.

This swapping operation is repeated until there exists no derivation of the form $G_{k(i-1)} \xRightarrow{fr_i @ fm_i} G_{ki} \xRightarrow{sr_{k+1} @ sm_{k+1}} G_{(k+1)i}$ with $k > i$, resulting in match consistent Derivation ③ $G_{00} \xRightarrow{sr_1} G_{10} \xRightarrow{sr_2} \dots \xRightarrow{sr_n} G_{n0} \xRightarrow{fr_1} G_{n1} \xRightarrow{fr_2} \dots \xRightarrow{fr_n} G_{nn}$ of source and forward rules depicted in Fig. 5.14.

COMPOSITION: Given match consistent Derivation ③ $G_{00} \xRightarrow{sr_1} G_{10} \xRightarrow{sr_2} \dots \xRightarrow{sr_n} G_{n0} \xRightarrow{fr_1} G_{n1} \xRightarrow{fr_2} \dots \xRightarrow{fr_n} G_{nn}$ of source and forward rules $\xRightarrow{\text{Lemma 3}}$

derivations $G_{(k-1)(i-1)} \xRightarrow{sr_k @ sm_k} G_{k(i-1)} \xRightarrow{fr_i @ fm_i} G_{kj}$ with $k > i$ are sequentially independent $\xRightarrow{\text{Thm. 5}}$ and can be replaced with $G_{(k-1)(i-1)} \xRightarrow{fr_i @ fm_i} G_{(k-1)i} \xRightarrow{sr_k @ sm_k} G_{ki}$ preserving match consistency and sequential independence.

This swapping operation is repeated until there exists no derivation of the form $G_{(k-1)(i-1)} \xRightarrow{sr_k @ sm_k} G_{k(i-1)} \xRightarrow{fr_i @ fm_i} G_{kj}$ with $k > i$, resulting in match consist-

ent Derivation ② depicted in Fig. 5.14 $\xrightarrow{\text{Lemma 2}}$ each match consistent derivation $G_{(i-1)(i-1)} \xRightarrow{sr_i} G_{i(i-1)} \xRightarrow{fr_i} G_{ii}$ in ② is E-related $\xrightarrow{\text{Thm. 4}}$ there is a synthesis construction leading to a direct derivation $G_{(i-1)(i-1)} \xRightarrow{r_i} G_{ii}$ where r_i is the E-concurrent rule $sr_i *_{\text{E}} fr_i$ according to Lemma 1. Repeating this synthesis construction for every derivation of the form $G_{(i-1)(i-1)} \xRightarrow{sr_i} G_{i(i-1)} \xRightarrow{fr_i} G_{ii}$ in ② results in Derivation ① depicted in Fig. 5.14. \square

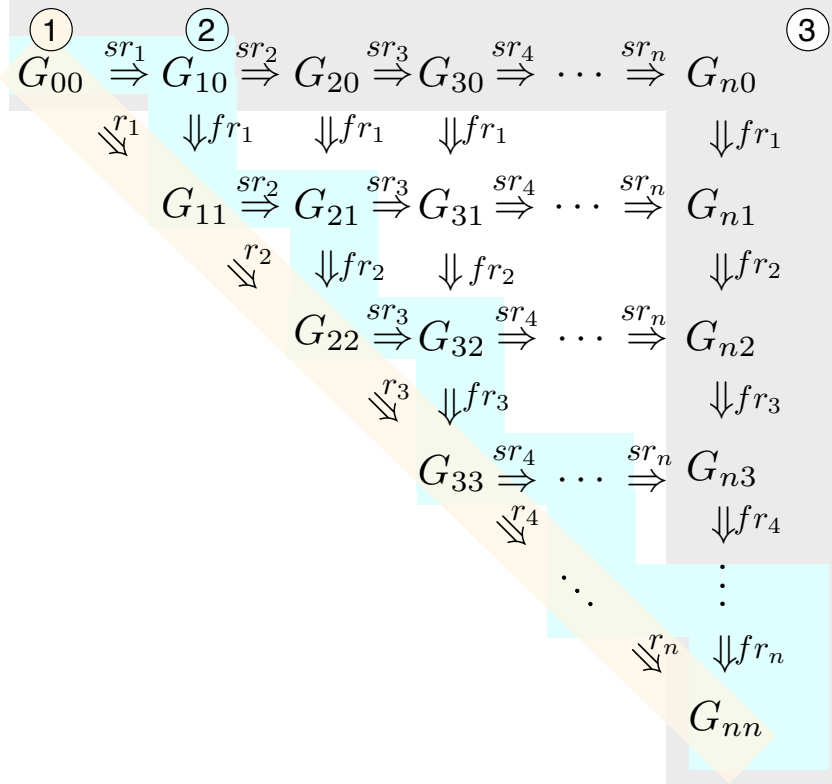


Figure 5.14: Derivations ①, ② and ③ used in the proof

The results of this section show how to operationalize TGG rules into source and forward rules, and guarantee that *every* consistent triple graph can be created by first applying a sequence of source rules, and then applying all necessary forward rules. This means that a forward synchronizer can basically propagate changes applied to the source component of a given triple graph by applying a series of forward rules until the triple graph is consistent. With the central concept of *match consistency* it is also clear how the forward rule applications must relate to the “applied” source rule applications.

5.2 PRECEDENCE-DRIVEN CONTROL ALGORITHM

The difficulty in model synchronization with TGGs lies in determining the sequence of source rule applications for the current source model. Once this is known then propagation is a simple matter of applying forward rules in a match consistent manner, i.e., controlled by the sequence of source rule applications.

Given a consistent triple graph $G = G_S \leftarrow G_C \rightarrow G_T$ and a source delta resulting in $G'_S \leftarrow G_C \rightarrow G_T$, the challenge is to determine *the* source rule derivation that can be extended to preserve *as much as possible* of the existing correspondence and target components of G . Although a simplistic backtracking algorithm could achieve this, it would (i) have exponential runtime and (ii) not be incremental, i.e., would not reuse previous results to speed up the process.

The following section presents the *precedence-driven* TGG-based synchronization algorithm of [71], which attempts to address exactly these challenges under certain conditions. The algorithm is extended appropriately to handle the new TGG features introduced in this thesis and, as the description in [71] was prior to an actual implementation, numerous improvements to the algorithm based on experience from establishing and using the current implementation in eMoflon are integrated into the current description.

The basic idea of [71] is to keep track of dependencies between elements in a given triple graph in such a way that this information can be updated *incrementally* when given a delta to be propagated. The dependencies are referred to as *precedences* and can be computed by determining match morphisms for TGG rules. If an element x is required in a certain match of a rule r as a context element for creating another element y , then x *precedes* y and y is thus dependent on x .

Maintaining a global view of all precedence relations in a triple graph is crucial as it is required to compute the consequences of *adding* new elements. This is difficult to accomplish using a simple translation protocol as *potential* dependencies (the added element has not been translated yet!) are required.

In the following and the rest of this thesis, all definitions and algorithms are presented for *forward* synchronization of a *source* delta, controlled using *source* precedences. As TGGs are symmetric in nature, this applies analogously to *backward* synchronization of *target* deltas. The examples provided, however, are sometimes in the forward direction, sometimes in the backward direction, depending on which is better to illustrate the current definition or algorithm.

Figure 5.15 provides a roadmap for the following Sect. 5.2.1 – 5.2.7, which introduce the precedence-driven synchronization algorithm.

The main algorithm SYNC (Alg. 9) applied for forward synchronization of a source delta, updates all source precedences, performs a forward translation of source elements controlled by these precedences, and finally reflects all changes in the target domain by updating all target precedences.

The update process is discussed in Sect. 5.2.1 – 5.2.3 and is performed primarily with the algorithm UPDATE (Alg. 6), which incrementally updates a *precedence graph*, a data structure used to track all precedences in a domain.

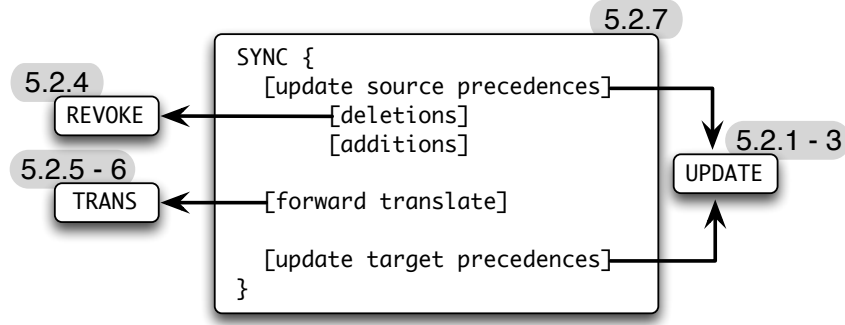


Figure 5.15: Roadmap for the following sections on precedence-driven synchronization

An additional data structure, a *translation protocol*, is introduced in Sect. 5.2.4 and is used by the algorithm REVOKE (Alg. 7) to handle deletions made by the user. After updating all source precedences and revoking deleted and affected source elements, all added as well as revoked but not deleted source elements are finally (re)translated using the algorithm TRANS (Alg. 8), presented together with corresponding restrictions in Sect. 5.2.5 - 5.2.6.

5.2.1 Restrictions for Precedence-Driven Synchronization

To be able to control the synchronization process using only precedence information, the usage of application conditions, already restricted for operationalization to source/target NACs, must be even further restricted. The idea is that if a rule is applicable with respect to precedence information, i.e., a match for all context elements exists, then NACs of the rule are only allowed to block rule application if a constraint over the type graph would be violated. As NACs are at least powerful enough to guarantee *negative constraints* [6], the following definition formalizes the class of *precedence-compatible* TGGs with appropriately restricted NAC usage.

Definition 52 (*Precedence-Compatibility*).

Let $TGG = (TG_S \xrightarrow{\sigma_{TG}} TG_C \xrightarrow{\tau_{TG}} TG_T, \mathcal{R})$ be an operationalizable triple graph grammar, \mathcal{R}_S the set of source rules for TGG with NACs, \mathcal{R}_S^- the set of source rules without NACs, $TGG_S = (TG, \mathcal{R}_S)$, $TGG_S^- = (TG, \mathcal{R}_S^-)$, and \mathcal{C}_{TG_S} a set of negative constraints for the source type graph TG_S .

TGG is *source precedence-compatible* if:

$$\forall G_S \in \mathcal{L}(TGG_S^-) \setminus \mathcal{L}(TGG_S), G_S \notin \mathcal{L}(TG_S, \mathcal{C}_{TG_S})$$

Target precedence-compatible TGGs are defined analogously.

TGG is *precedence-compatible* if it is source and target precedence-compatible.

After a series of synchronization steps, it must be guaranteed that only *valid* triple graphs, i.e., triple graphs that do not violate any constraints, are produced by the synchronizer. This is formalized in the following as *schema-compliance* meaning that the NACs used in a TGG are *sufficient* for this purpose. In this light, Def. 52 requires that the NACs in a TGG be *necessary* for schema-compliance.

Definition 53 (*Schema-Compliance*).

Let $TGG = (TG_S \xrightarrow{\sigma_{TG}} TG_C \xrightarrow{\tau_{TG}} TG_T, \mathcal{R})$ be an operationalizable triple graph grammar, \mathcal{R}_S the set of source rules for TGG, $TGG_S = (TG, \mathcal{R}_S)$, and \mathcal{C}_{TG_S} a set of negative constraints for the source type graph TG_S .

$G_S \in \mathcal{L}(TG_S)$ is *schema-compliant* if $G_S \in \mathcal{L}(TG_S, \mathcal{C}_{TG_S})$.

TGG_S is *source schema-compliant* if $\mathcal{L}(TGG_S) \subseteq \mathcal{L}(TG_S, \mathcal{C}_{TG_S})$.

Target schema-compliance is defined analogously.

TGG is *schema-compliant* if it is source and target schema-compliant.

All possible combinations of Def. 52 and 53 are depicted in Fig. 5.16, showing in each case the language $\mathcal{L}(TG_S)$ of all typed source graphs, the language $\mathcal{L}(TG_S, \mathcal{C}_{TG_S})$ of all schema-compliant source graphs, the language $\mathcal{L}(TGG_S^-)$ of triple graphs generated by rules of the TGG without source NACs, and finally, the language $\mathcal{L}(TGG_S)$ generated by rules of the TGG using source NACs to prevent violations of source constraints.

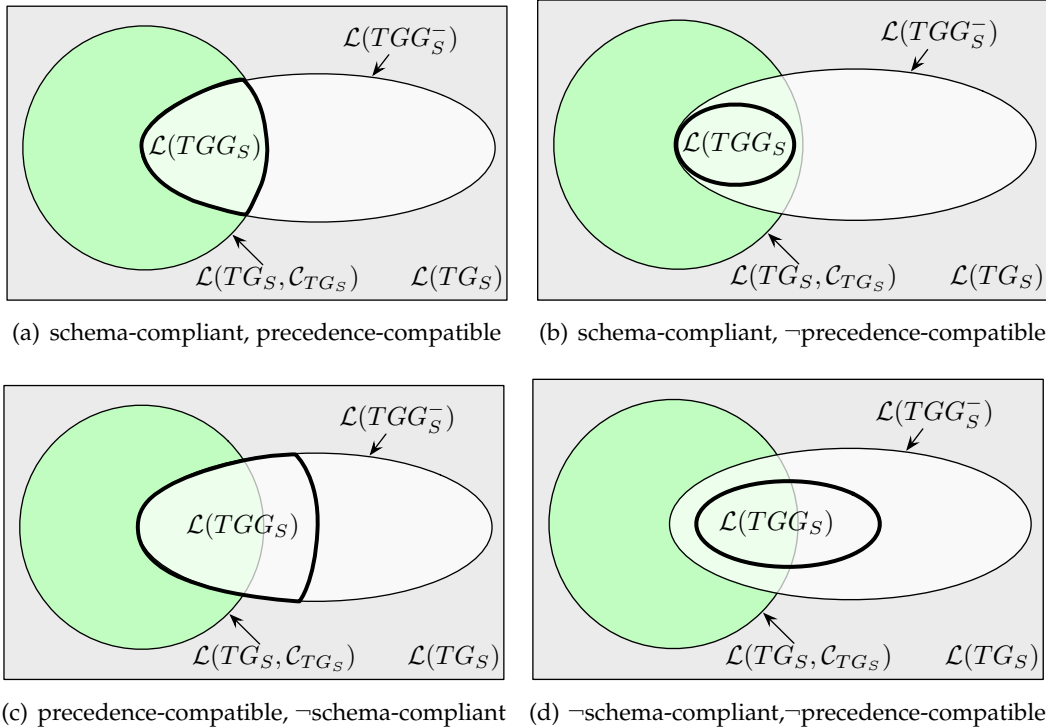


Figure 5.16: Precedence-compatibility and schema-compliance

Example 33 (*Precedence-Compatibility and Schema-Compliance*).

To illustrate precedence-compatibility and schema-compliance with a concrete example, we shall revisit Ex. 20 in a slightly simplified version. The rules of the TGG can be organized into two groups:

1. An axiom creating the basic container structures of source and target models, and two Goto rules that handle the creation of Goto nodes and corresponding Commands. The first version of these rules is depicted in Fig. 5.17. Note that attribute conditions are used in `PaintGotoRule` to ensure that the newly created Goto node is a direct sibling, with respect to indices, of the previous paint node in the tree.
2. Paint rules that create a new paint node after an existing Goto node in the tree. Recall that paint nodes can be *modal*, meaning that they can be omitted in the CLS textual concrete syntax (the label of the colour node in the tree is set by the parser to the empty string) if the previous colour is to be used. In such a case, the colour attribute of the created PAINT element in the model must be set with the last non-trivial colour node in the tree, determined by searching backwards from the current position using a source dynamic condition. The first version of both paint rules is depicted in Fig. 5.18. Note that this is a freedom of choice – paint nodes can but do not *have to* be modal.

Version 0 of the TGG (whose rules are depicted in Fig. 5.17 and Fig. 5.18) is trivially precedence-compatible *and* schema-compliant as no constraints have been specified. Although this is ideal for precedence-driven synchronization, this is almost never the case in practice. Even for our simple example, the TGG already produces branches of Commands, which are meaningless as we only want a *linear* sequence of CLS operations.

These and other requirements are formalized with the set of negative source and target constraints depicted in Fig. 5.19 and Fig. 5.20. Target constraints `NoMultipleOutgoingNext` and `NoMultipleIncomingNext` ensure that only linear sequences of connected Commands are created, while `NoMultiplePaint` enforces that every Command has at *most* one PAINT.

As these target constraints do not yet ensure that only *contiguous* chains of Commands are created, the source constraint `NoMultipleFirstGoto` forbids multiple Goto nodes with `index == 0`. Together with attribute conditions to enforce a monotonously increasing index in the tree, this guarantees that only a single contiguous chain of Commands can be created by the TGG.

Note that not all well-formedness conditions can be specified using only negative constraints. It is, for example, impossible to ensure that every Command has *at least* one PAINT; this would be a positive constraint (cf. Def. 9). Indeed, supporting a larger class of conditions is crucial future work.

Taking these four negative constraints into account, Version 0 of our TGG is no longer schema-compliant as it produces triple graphs that violate these constraints. To make the TGG schema-compliant, an experienced TGG developer can specify appropriate source and target NACs for each rule. Version 1 of the axiom and Goto rules is depicted in Fig. 5.21.

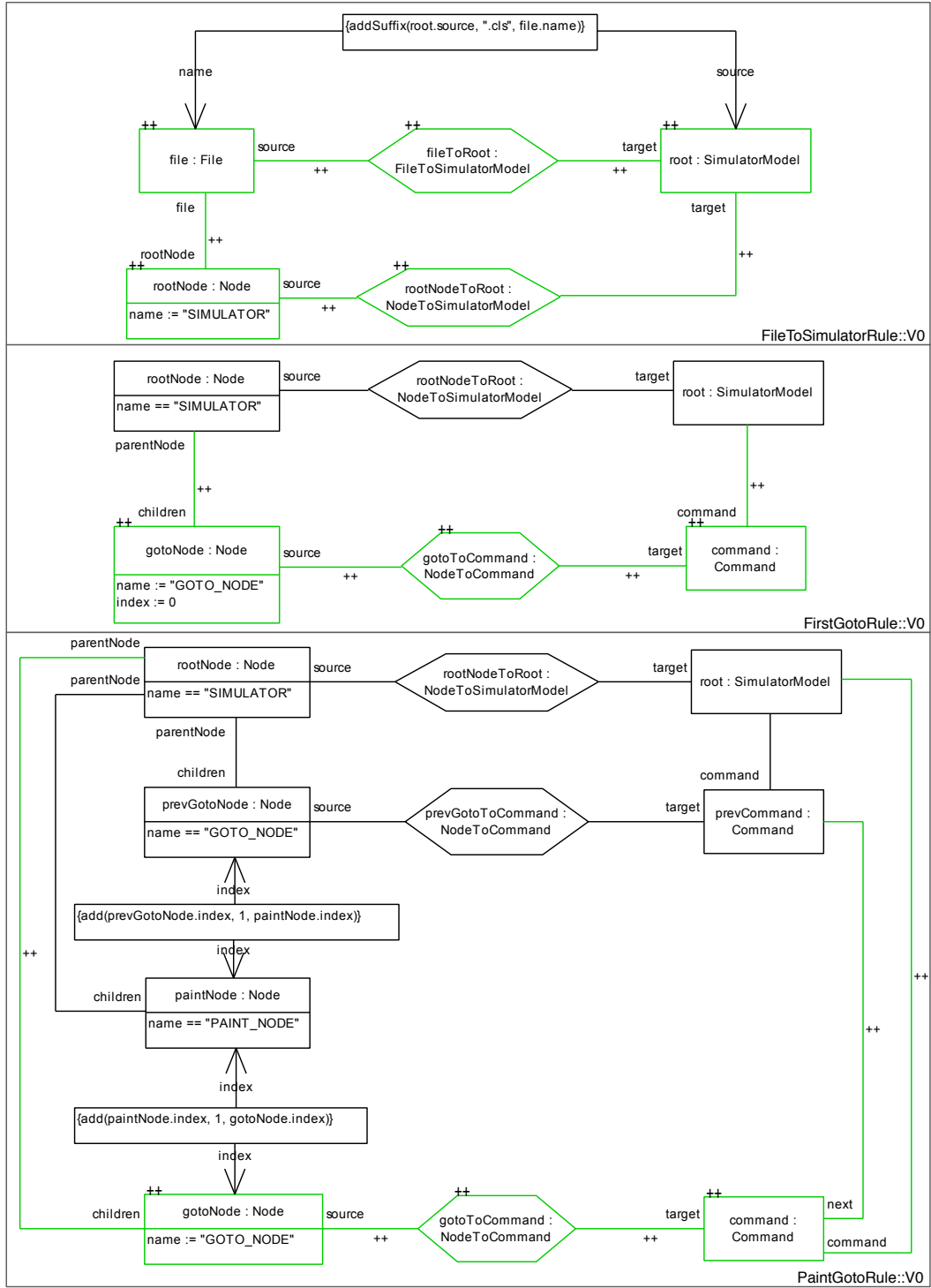


Figure 5.17: Axiom and Goto rules

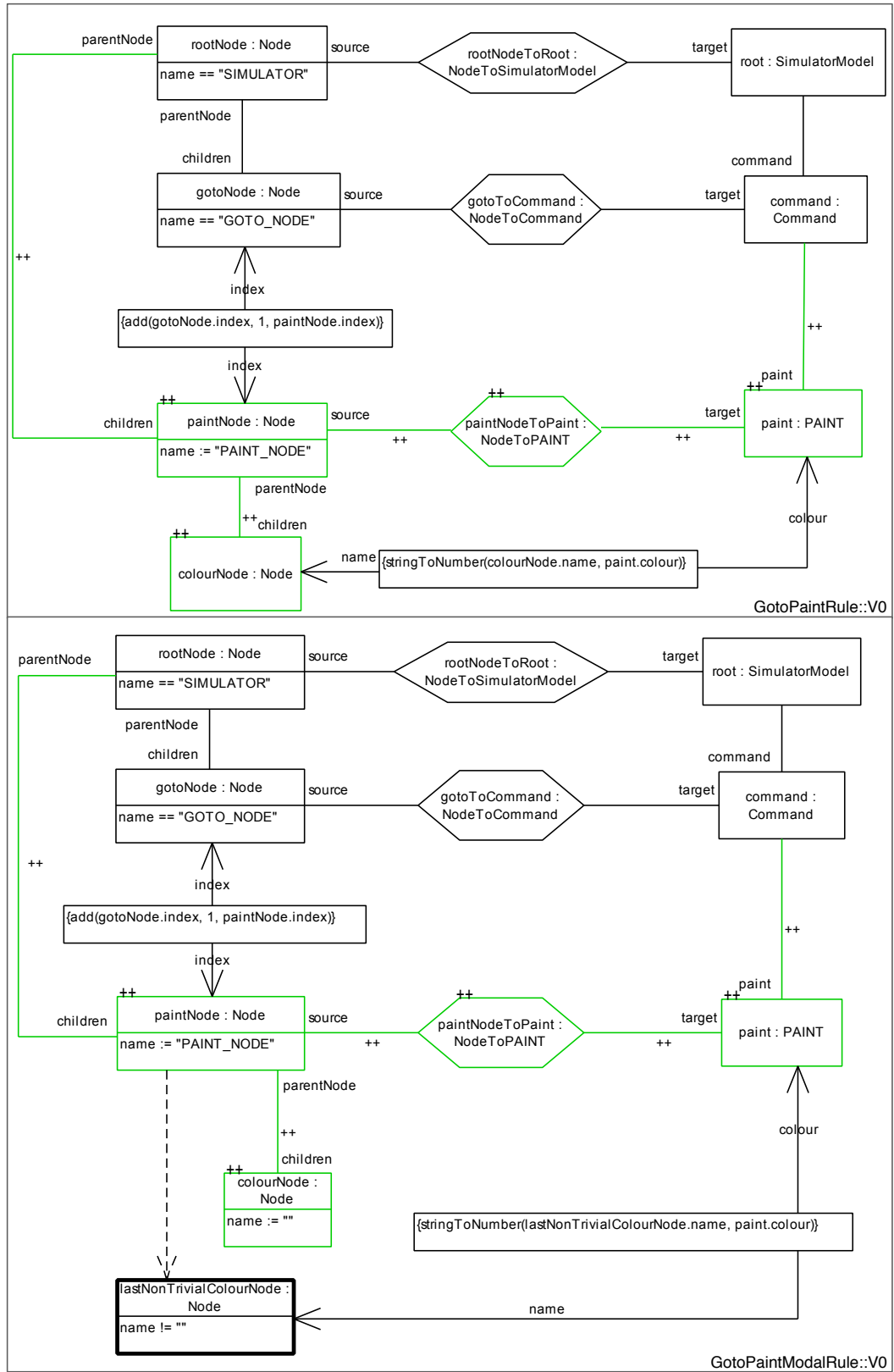


Figure 5.18: Paint rules

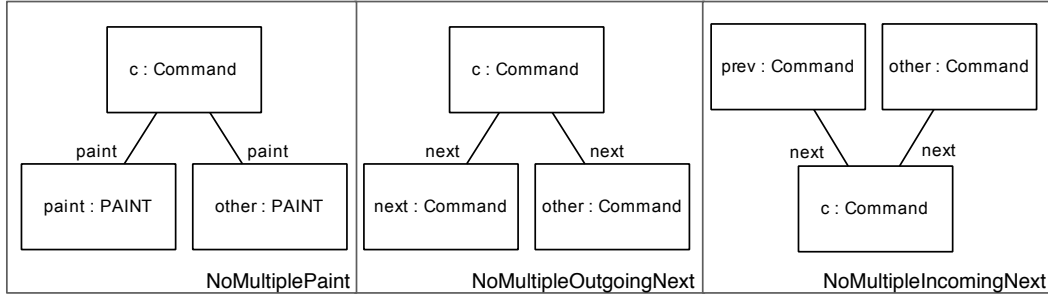


Figure 5.19: Target negative constraints

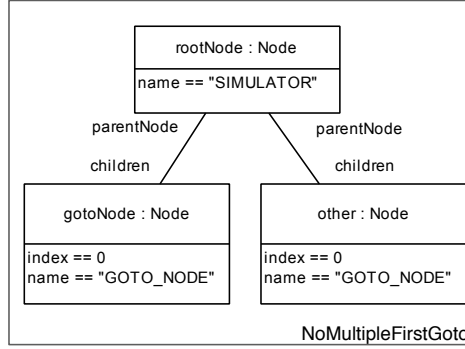


Figure 5.20: Source negative constraint

As FileToSimulatorRule is an axiom, i.e., only creates elements, it is impossible to prevent it from producing any new constraint violations safe from completely blocking rule application in general. In this case we are fortunate and FileToSimulatorRule does not violate any of the constraints.

FirstGotoRule would violate NoMultipleFirstGoto if applied to a rootNode and root that already contain elements. The NACs added in Version 1 of the rule appear to solve this problem but we shall revisit this in a moment when we discuss if the TGG is now schema-compliant or not.

For PaintGotoRule, it is necessary to ensure that the new Command is only added to an existing Command that does not already have a next Command. Analogously, Version 1 of the Paint rules depicted in Fig. 5.22 shows NACs preventing the creation of PAINTs when the containing Command already has one.

Given TGG rules and a set of source and target negative constraints, it is a challenging task to specify the exact set of sufficient and necessary NACs so that the TGG becomes both precedence-compatible and schema-compliant. Even experienced TGG developers easily make mistakes – in this case Version 1 of the TGG (whose rules are depicted in Fig. 5.21 and Fig. 5.22) is now schema-compliant but is no longer precedence-compatible. The mistake here was adding a target NAC to FirstGotoRule that is too strong in the following sense: with respect to the target language there is no corresponding negative constraint for the target NAC. This means that there exist target graphs $G_T \in \mathcal{L}(\text{TGG}_T^-) \setminus \mathcal{L}(\text{TGG}_T)$ that do not violate any target constraints, i.e., $G_T \in \mathcal{L}(\text{TG}_T, \mathcal{C}_{\text{TGG}_T^-})$. The TGG is, consequently, not target precedence-compatible as the NAC is *not* necessary for ensuring *target* schema-compliance!

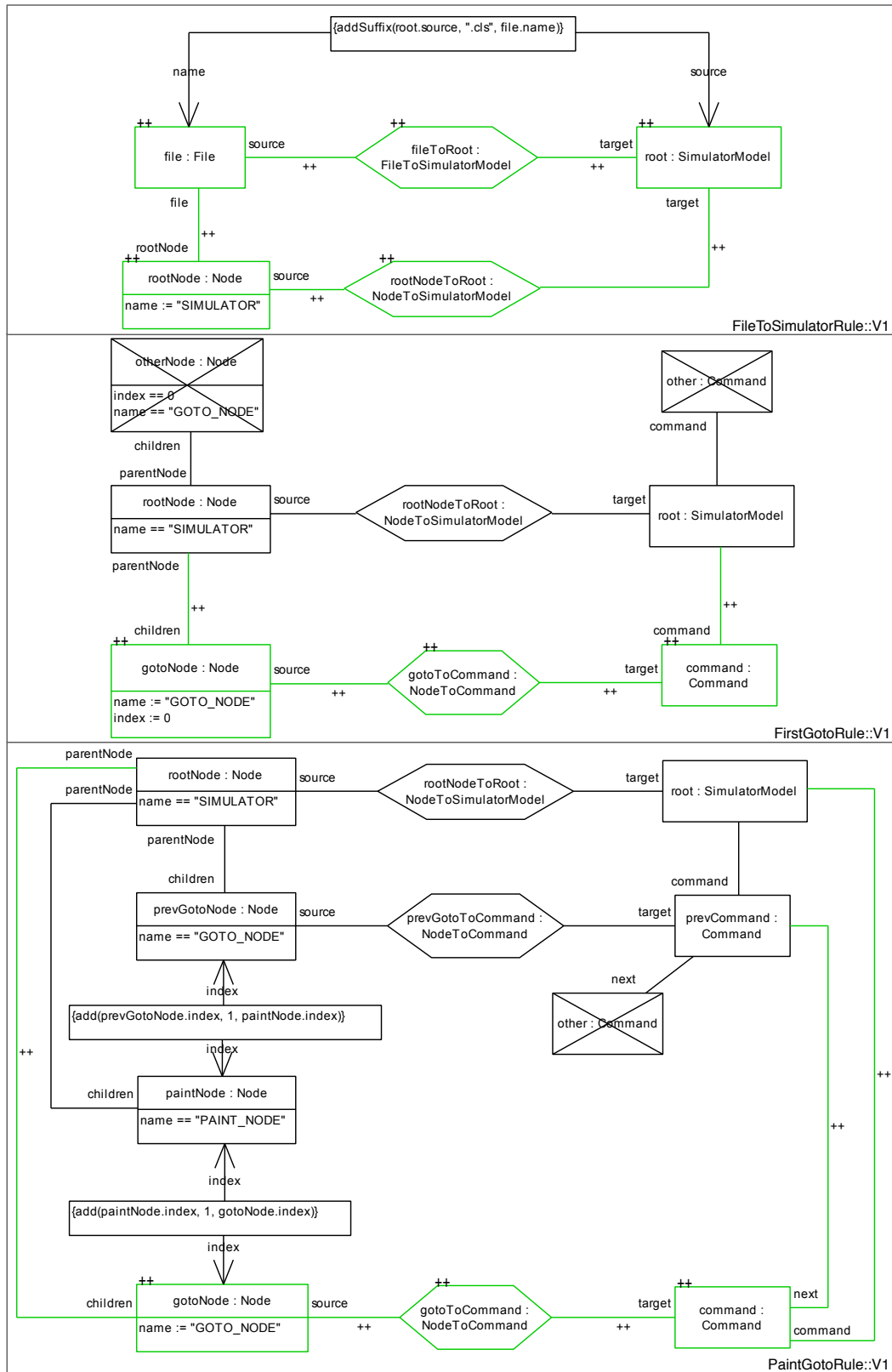


Figure 5.21: Axiom and Goto rules with NACs

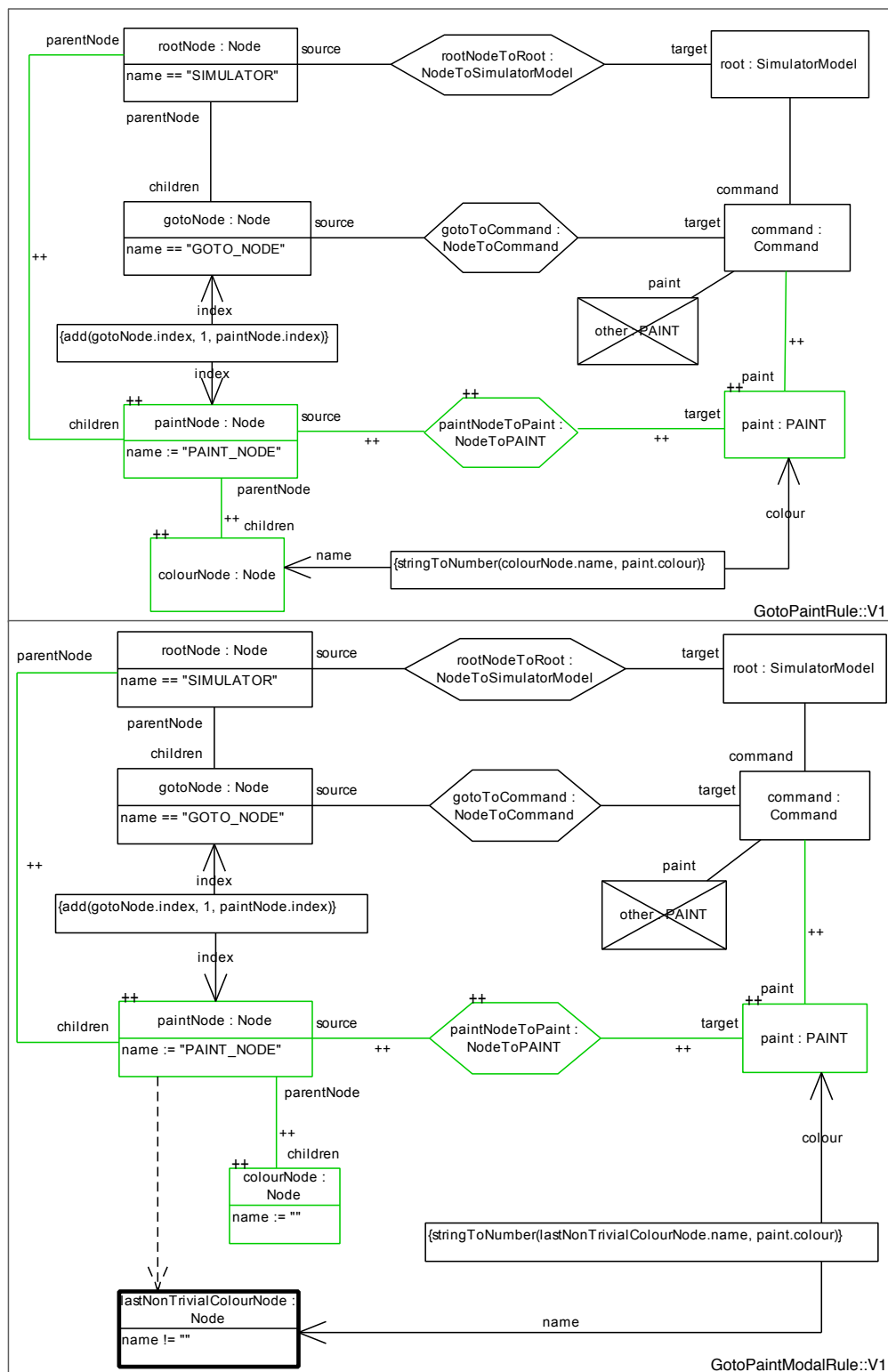


Figure 5.22: Paint rules with NACs

The corrected version of FirstGotoRule is depicted in Fig. 5.23. The remaining source NAC used to ensure schema-compliance corresponds directly to a negative source constraint. The TGG with rules in Version 1 and with this corrected version of FirstGotoRule is now schema-compliant and precedence-compatible as required. In Chapter 6, a construction technique will be presented to automate this tedious and error-prone specification of necessary and sufficient NACs for schema-compliance.

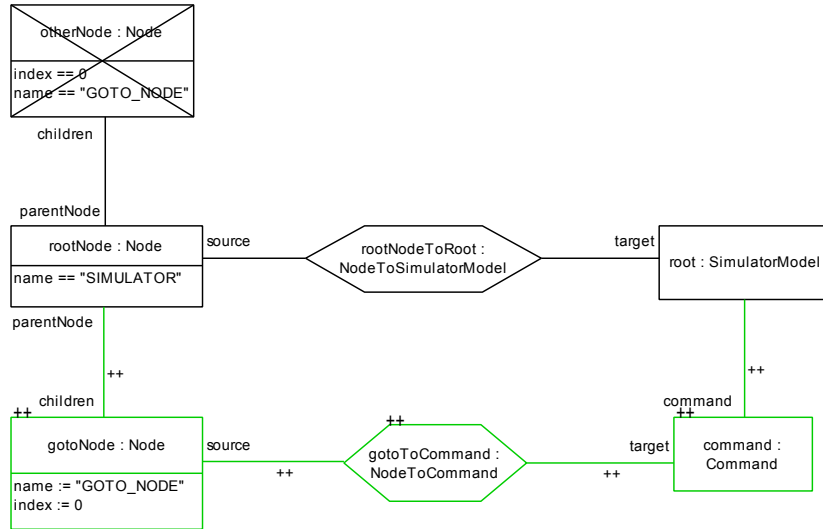


Figure 5.23: Corrected FirstGotoRule

5.2.2 Collecting Dependencies with Precedence Matches

When propagating a source delta, the final source graph is already given and the precedence (dependency) relations must be updated to take the newly added elements into account. This means that instead of applying source rules (the elements are already present in the source graph and do not need to be re-created), source rule derivations are *induced* by determining co-matches of source rules in the existing source graph. The advantage of such *source precedence matches*, formalized in the following, is that precedence relations can be updated *incrementally* by simply collecting all possible matches for all added elements.

Definition 54 (*Source Precedence Match Morphism and Induced Source Derivation*).

Let TGG be a precedence-compatible triple graph grammar, \mathcal{R}_S the set of source rules for TGG, $sr : SL \rightarrow SR \in \mathcal{R}_S$ a source rule, and $G_S \in \mathcal{L}(TG_S, \mathcal{C}_{TG_S})$ a schema-compliant source input graph.

The set of *precedence conditions* for sr is defined as:

$$\mathcal{PC}_S(sr) := \{\mathcal{N}_{DEC}, \mathcal{DC}_S^{pre}, \mathcal{DC}_S^{post}, \mathcal{C}_{A(X)}\}$$

where $\mathcal{N}_{DEC} := \text{determineDEC}(sr, \mathcal{R}_S, TG_S)$ is a set of *dangling edge conditions* for sr (cf. Alg. 5), $\mathcal{DC}_S^{pre}, \mathcal{DC}_S^{post}$ are the sets of dynamic pre- and postconditions for sr , respectively, and $\mathcal{C}_{A(X)}$ is the set of attribute conditions for sr .

A morphism $sm' : SR \rightarrow G_S \in \mathcal{M}$ is a *source precedence match morphism* if it fulfils all *precedence conditions* $\mathcal{PC}_S(sr)$ for sr , i.e., $sm' \models \mathcal{PC}_S(sr)$.

For a source precedence match morphism $sm' : SR \rightarrow G_S$, its *induced source derivation* is defined as $v := sm(SL) \xrightarrow{sr @ \overline{sm}} sm'(SR)$, where (cf. Fig. 5.24):

1. The match morphism $sm' : SR \rightarrow G_S$ is decomposed into an epimorphism (surjective on nodes and edges) $\overline{sm'} : SR \rightarrow sm'(SR)$, and an inclusion $sm'(SR) \subseteq G_S$ (this is possible according to [6]). As $sm' \in \mathcal{M}$, this means that $\overline{sm'}$ is an isomorphism (injective *and* surjective on nodes and edges).
2. The morphisms $\overline{sm} : SL \rightarrow sm(SL), sr' : sm(SL) \rightarrow sm'(SR)$ are constructed as a pushout complement in Fig. 5.24, which exists as $\overline{sm'}$ is an isomorphism (the so-called “gluing condition” guaranteeing this is trivially fulfilled [28]).

The following relations are defined for the corresponding induced source derivation v of the precedence match morphism sm' with fulfilled dynamic conditions $(i_k, \beta_k) \in \{\mathcal{DC}_S^{pre} \cup \mathcal{DC}_S^{post}\}, i_k : I_k \rightarrow SR, \beta_k(sm') : I_k \rightarrow C_k, q_k : C_k \rightarrow G_S$:

$$\begin{aligned} \text{created}(v) &:= sm'(SR) \setminus sm(SL) \\ \text{context}_{DC} &:= [\bigcup_k q_k(C_k)] \setminus \text{created}(v) \text{ (cf. Fig. 5.25)} \\ \text{context}(v) &:= sm(SL) \cup \text{context}_{DC} \end{aligned}$$

Target precedence match morphisms and induced target derivations are defined analogously.

Intuitively, given a source rule $sr : SL \rightarrow SR$, a source precedence match morphism $sm' : SR \rightarrow G_S$ can be used to identify elements in an existing graph G_S that could have been (1) used as context for applying the source rule, and (2) could have been created by applying the source rule. These induced precedence relations are, however, only *potential* dependencies, as there could be multiple source precedence match morphisms for different rules with overlapping elements, i.e., when a new element is added to a source graph, there might be multiple ways of propagating this change using different rules.

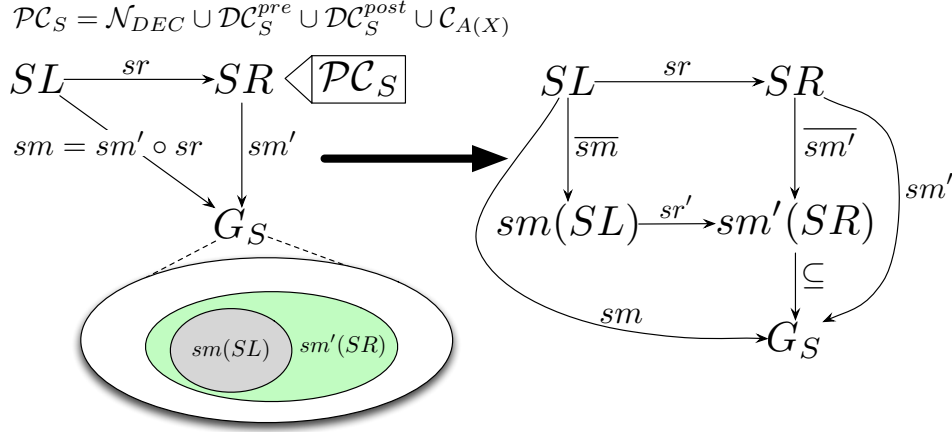


Figure 5.24: Source precedence match and induced source derivation

From the source precedence match $sm'(SR)$ and the source rule, it is clear which elements would be created by the corresponding, *induced*, source derivation. Determining all context elements from the match is slightly more complex as source dynamic conditions must be taken into account. The elements context_{DC} required to fulfil dynamic conditions must be already present in the graph, and are thus part of the context of the induced source derivation. This is depicted visually in Fig. 5.25.

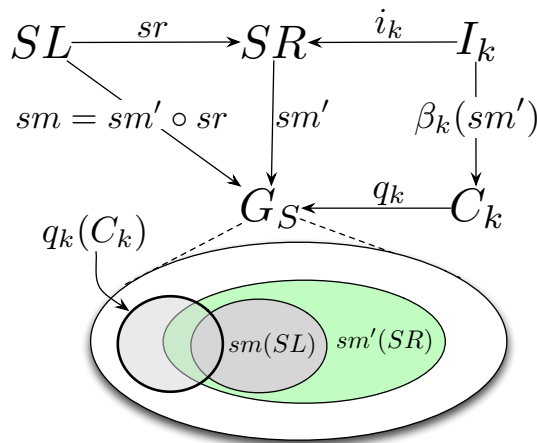


Figure 5.25: Additional context due to fulfilled dynamic conditions

An important point to note here is that all source NACs are omitted when determining a source precedence match. This is only possible because G_S is required to be schema-compliant, and the TGG to be precedence-compatible. Under these two conditions we know that the omitted source NACs would only block constraint violations and, as G_S is schema-compliant and TGG rules are monotonic creating, there cannot have been any constraint violations so all source NACs can be safely removed. Being able to ignore all source NACs reduces the complexity of collecting new precedence relations as matches can be determined without keeping track of which elements are already “present” or not, and how the source NACs are to be correctly interpreted for existing elements. Without this simplification, one would either have to re-create the newly added elements or “mark” the elements in some way. This gets complex as there can be multiple possibilities using different rules and matches. A further practical but equally important point is efficiency – checking all constraints once for G_S is more efficient than constantly checking all NACs for all matches of all rules.

In some cases, a source derivation might be induced that could never be part of a source derivation for the given source graph G_S . This leads to false precedences and should be avoided. To this end, a set of NACs implementing a *Dangling Edge Condition* (DEC) are generated using Alg. 5 for each source rule and are added to the set of precedence conditions \mathcal{PC}_S , which must be fulfilled by source precedence matches (Fig. 5.24). The main idea is to integrate a look-ahead as a condition that checks if a “dangling edge”, i.e., an edge that would not be translatable with any other rule, would be left after applying the induced source derivation. If this is the case then the precedence match is invalid and is discarded.

Definition 55 (*Shorthand Notation for Readability*).

The following shorthand notations are used in the rest of this thesis:

$$\begin{aligned} e : \text{src}(e) = v, \text{trg}(e) = v' &\Leftrightarrow e : v \rightarrow v' \\ (e : v \rightarrow v') \vee (e : v \leftarrow v') &\Leftrightarrow e : v - v' \end{aligned}$$

Algorithm 5 loops through all nodes created by a source rule on Line 3. For each node n , all possible types of edges that could be connected to the node according to the type graph are determined on Line 4. For each of these potentially incident edges e , a rule sr^* is demanded on Lines 5–6 that is able to translate it with n as context. If no such rule exists, then e can never be translated and is thus a “dangling edge”. This situation is forbidden with the NAC N , created and added to the set of DEC NACs on Lines 7 – 8.

Algorithm 5 Determining dangling edge conditions \mathcal{N}_{DEC} for source rule sr

Require:

- (a) TG_S : the source type graph
- (b) \mathcal{R}_S a set of source rules
- (c) $\text{sr} : \text{SL} \rightarrow \text{SR} \in \mathcal{R}_S, \text{SL}, \text{SR} \in \mathcal{L}(\text{TG}_S)$: a source rule

```

1: procedure DETERMINEDEC( $\text{sr}, \mathcal{R}_S, \text{TG}_S$ ):  $\mathcal{N}_{\text{DEC}}$ 
2:    $\mathcal{N}_{\text{DEC}} \leftarrow \emptyset$ 
3:   for all  $n \in V_{\text{SR}} \setminus V_{\text{SL}}$  do
4:     for all  $\bar{e} : \bar{n} - \bar{m} \in E_{\text{TG}_S}, \bar{n}, \bar{m} \in V_{\text{TG}_S}, \bar{n} = \text{type}(n)$  do
5:       if  $\nexists \text{sr}^* : \text{SL}^* \rightarrow \text{SR}^* \in \mathcal{R}_S, \exists e^* : n^* - m^* \in E_{\text{SR}^*} \setminus E_{\text{SL}^*},$ 
6:          $\text{type}(e^*) = \bar{e}, n^* \in V_{\text{SL}^*}, \text{type}(n^*) = \bar{n}, m^* \in V_{\text{SR}^*}$  then
7:          $N \leftarrow (V_{\text{SR}} \cup \{m : \text{type}(m) = \bar{m}\}, E_{\text{SR}} \cup \{e : n - m\})$ 
8:          $\mathcal{N}_{\text{DEC}} \leftarrow \mathcal{N}_{\text{DEC}} \cup N$ 
9:       end if
10:    end for
11:  end for
12:  return  $\mathcal{N}_{\text{DEC}}$ 
13: end procedure

```

Example 34 (*Determining DEC NACs for filtering source precedence matches*). ———

To illustrate DEC NACs and how invalid precedence matches are filtered with a concrete example, Fig. 5.26 depicts the target rule for FirstGotoRule with DEC NACs generated using Alg. 5. In practice, the set of DEC NACs is further reduced using information from multiplicities, inheritance, composition, and any other constraints in the respective metamodel.

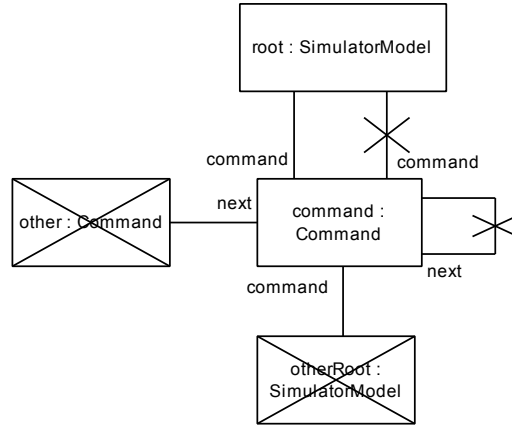


Figure 5.26: FirstGotoTargetRule with DEC NACs

Four dangling edges are identified: a second command link to root, a command link to a different SimulatorModel otherRoot, and incoming next links from the same Command object command and from another Command object other. If command is created using FirstGotoRule, these edges could never exist in a consistent target graph.

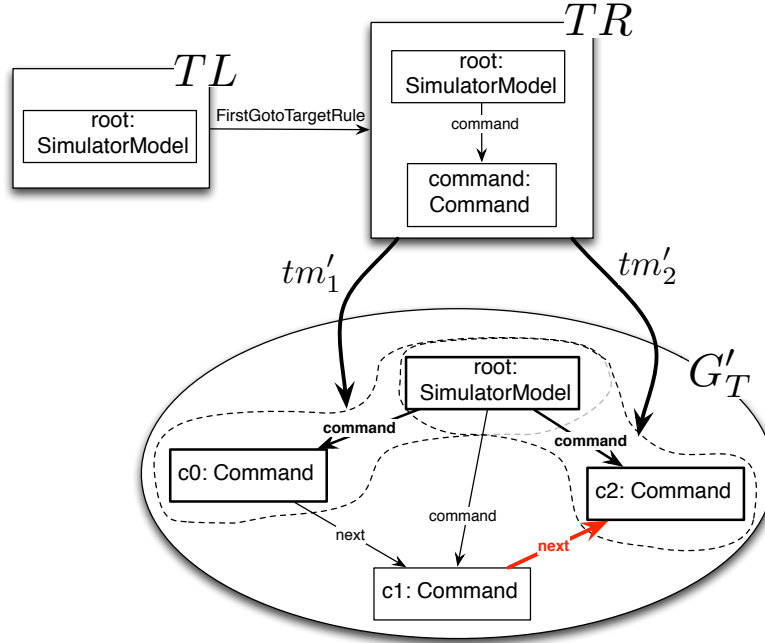


Figure 5.27: An invalid target precedence match filtered out using a DEC NAC

The only interesting case is the forbidden incoming next link from another Command object. Figure 5.27 depicts a valid target precedence match morphism tm'_1 as well as an invalid target precedence match morphism tm'_2 , which results in a dangling incoming next link and is thus filtered out by the corresponding DEC NAC (Fig. 5.26). The point here is that no target rule creates a new next incoming link for an existing Command.

5.2.3 Tracking Dependencies Incrementally with a Precedence Graph

We are now ready to define the data structure, referred to as a *precedence graph*, used to hold all information concerning identified precedence relations between the elements² of a source/target graph. Note that Def. 56 characterizes a precedence graph without stipulating in anyway how such a data structure can be efficiently constructed, and incrementally updated.

Definition 56 (*Source Precedence Graph*).

Let $TGG = (TG_S \xrightarrow{\sigma_{TG}} TG_C \xrightarrow{\tau_{TG}} TG_T, \mathcal{R})$ be an operationalizable triple graph grammar, \mathcal{R}_S the set of source rules of TGG, and $G_S \in TG_S$ a source graph.

A *source precedence graph* is a graph $PG_S = (V_{PG_S}, E_{PG_S})$ such that:

1. Each node $v \in V_{PG_S}$ is a direct derivation $H \xrightarrow{sr@sm} H'$ with a source rule $sr \in \mathcal{R}_S$ and $H \subset H' \subseteq G_S$.
2. $\forall (v_i, v_j) \in V_{PG_S}, \text{created}(v_i) \cap \text{context}(v_j) \neq \emptyset \Leftrightarrow \exists e : v_i \rightarrow v_j \in E_{PG_S}$.
3. For every $\emptyset = G_{S_0} \xrightarrow{sr_1@sm_1} G_{S_1} \xrightarrow{sr_2@sm_2} \dots \xrightarrow{sr_n@sm_n} G_{S_n} = G_S$ with source rules in \mathcal{R}_S , there exist corresponding direct derivations v_1, v_2, \dots, v_n in V_{PG_S} such that $\forall i \in \{1, \dots, n\}, G_{S_i} \xrightarrow{sr_i@sm_i} G_{S_{i+1}}$, the corresponding derivation v_i is of the form $H_i \xrightarrow{sr_i@sm_i} H_{i+1}$ with $H_i \subseteq G_{S_i}$ and $H_{i+1} \subseteq G_{S_{i+1}}$.

Target precedence graphs are defined analogously.

Intuitively, a precedence graph captures all *possible* ways of creating all elements in a graph, together with induced context dependencies.

Condition (1) in Def. 56 enforces correctness of the precedence graph, i.e., no “invalid” nodes are allowed that do not correspond to source rule derivations.

Condition (2) handles dependencies, demanding on the one hand that the precedence graph capture every context dependency with an edge, and on the other hand that only such edges be present.

Finally, Condition (3) enforces completeness of the precedence graph, i.e., that every possible derivation of the graph can be traced as a sequence of nodes connected appropriately in the precedence graph.

Example 35 (*A target precedence graph for the running example*). —————

Figure 5.28 depicts a target graph G_T and a corresponding target precedence graph PG_T for G_T . G_T consists of a `SimulatorModel` root and two connected `Commands` `c0` and `c1`. PG_T is visualized as a graph with nodes and edges, where each node contains the elements created by the induced target rule derivation and is labelled with the name of the corresponding TGG rule.

² Note that

As there is only one way of creating a `SimulatorModel`, there is also a single node ① in the precedence graph. `FileToSimulatorTargetRule` is an axiom and, therefore, ① has no incoming edges.

Although there are two target rules that create `Commands`, only `FirstGotoTargetRule` creates a `Command` without demanding another `Command` as context. There is, therefore, also only a single node ② for creating `c0` and the command link connecting it to root. This link is the reason why there is an edge connecting ① with ②.

The second `Command` `c1` cannot be created using `FirstGotoTargetRule` as it is blocked by a DEC NAC (cf. Fig. 5.26). Node ③ requires context elements from both ① and ②, which is why it has two incoming edges.

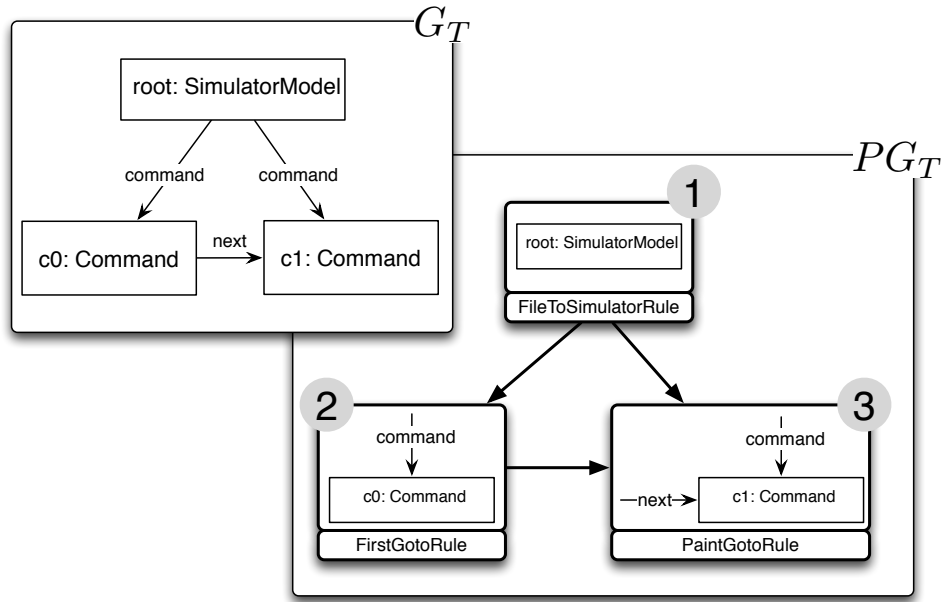


Figure 5.28: Target precedence graph PG_T for target graph G_T

Intuitively, we can already guess that ③ is problematic; in the actual TGG rule `PaintGotoRule::V1`, a paint node is required as context in the source domain and can only be present if `prevCommand` has a `PAINT`. As `c0` certainly has no `PAINT` in this case, `PaintGotoRule` can never be actually applied to translate G_T , even though this is implied by PG_T . We shall revisit this “problem”, define an additional property for TGG rules, and correct `PaintGotoRule::V1` later in this chapter (cf. Fig. 5.37).

The reason for introducing precedence matches and precedence graphs is to be able to update a precedence graph with newly added elements in an *incremental* manner. After introducing some relations on precedence graphs to improve readability in the following definition, Alg. 6 states how a precedence graph can be updated given a set of new elements for which precedences are to be determined.

Definition 57 (*Relations on Precedence Graphs and Precedence Matches*).

Given a precedence graph PG_S , and a node $v \in V_{PG}$, the following relations are defined for v as follows:

$$\begin{aligned} \text{siblings}(v) &:= \{v' \in V_{PG} \mid \text{created}(v) \cap \text{created}(v') \neq \emptyset\} \\ \text{children}(v) &:= \{v' \in V_{PG} \mid \exists e \in E_{PG} : v \rightarrow v'\} \\ \text{parents}(v) &:= \{v' \in V_{PG} \mid \exists e \in E_{PG} : v' \rightarrow v\} \\ \overline{\text{children}}(v) &:= \text{transitive closure over } \text{children}(v) \end{aligned}$$

The corresponding sets of nodes in each relation are depicted visually in Fig. 5.29.

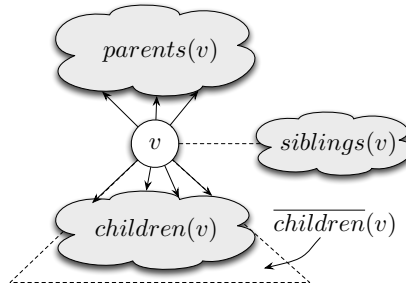


Figure 5.29: Relations on precedence graphs and precedence matches used in Alg. 6

The procedure update (Alg. 6) starts by collecting all derivations for the new elements in the given set X on Line 3. The auxiliary procedure `collectDerivations` determines all source precedence matches and adds the induced source derivations to the set V_X .

After these new nodes are added to the precedence graph, a second auxiliary procedure `remove` is used to transitively remove all siblings of newly added nodes and their dependencies from the precedence graph. This step is counter-intuitive but is necessary to avoid invalidating existing matches by adding new elements. This will be explained in detail with a concrete example and in the proof of Lemma 4.

Finally, the removed elements are integrated back into the precedence graph by collecting derivations for them on Line 8, and adding edges representing context dependencies using the auxiliary procedure `calcDeps` on Line 10.

To determine all derivations for a given set X of elements, `collectDerivations` determines all precedence matches sm' for an element $x \in X$. If the match fulfils all precedence conditions, then its induced derivation is collected.

To remove a node v from a precedence graph, `remove` first of all determines all its children and siblings, i.e., all nodes that depend on it, and all nodes that co-create at least one element with v , respectively. The node is then deleted from the graph (Lines 4–5) and `remove` recurses for all children and siblings (Line 8 and Line 11). Finally, the set of “removed” elements, i.e., elements for which all creating nodes have been thus removed from the precedence graph, is updated (Line 14) and returned.

Algorithm 6 Updating a source PG (analogously for target PGs)

Require:

- (a) $TGG = (TG, \mathcal{R})$: operationalizable, source precedence-compatible triple graph grammar.
 - (b) \mathcal{R}_S^- : set of source rules without NACs for TGG , $TGG_S^- = (TG, \mathcal{R}_S^-)$.
 - (c) $G_S \in \mathcal{L}(TG_S, \mathcal{C}_{TG_S})$, $G_S \in \mathcal{L}(TGG_S^-)$: input source graph.
 - (d) $X \subseteq G_S$: elements to be integrated into PG_S , the precedence graph for $G_S \setminus X$.
- 1: **procedure** $UPDATE(X, G_S, PG_S, \mathcal{R}_S^-)$: (X', PG'_S)
 - 2: $X' \leftarrow \emptyset, PG'_S \leftarrow PG_S$
 - 3: $V_X \leftarrow \text{COLLECTDERIVATIONS}(X, G_S, \mathcal{R}_S^-)$
 - 4: $V_{PG'_S} \leftarrow V_{PG_S} \cup V_X$
 - 5: **for all** $v \in V_X$ **do**
 - 6: $(X', PG'_S) \leftarrow \text{REMOVE}(v, X', PG'_S)$
 - 7: **end for**
 - 8: $V_{X'} \leftarrow \text{COLLECTDERIVATIONS}(X' \setminus X, G_S, \mathcal{R}_S^-)$
 - 9: $V_{PG'_S} \leftarrow V_{PG'_S} \cup V_{X'} \cup V_X$
 - 10: $PG'_S \leftarrow \text{CALCDEPS}(PG_S, PG'_S)$
 - 11: **return** (X', PG'_S)
 - 12: **end procedure**
-

- 1: **procedure** $\text{COLLECTDERIVATIONS}(X, G_S, \mathcal{R}_S^-)$: V_X
 - 2: $V_X \leftarrow \emptyset$
 - 3: **for all** $x \in X$ **do**
 - 4: **for all** $sr : SL \rightarrow SR \in \mathcal{R}_S^-, sm' : SR \rightarrow G_S, sm = sm' \circ sr,$
 - 5: $x \in \text{created}(sm') \cup \text{context}(sm'),$
 - 6: $sm' \models \mathcal{PC}_S(sr)$ (cf. Def. 54) **do**
 - 7: $V_X \leftarrow V_X \cup [sm(SL) \xrightarrow{sr@sm} sm'(SR)]$
 - 8: **end for**
 - 9: **end for**
 - 10: **return** V_X
 - 11: **end procedure**
-

```

1: procedure REMOVE( $v, X, PG_S$ ): ( $X', PG'_S$ )
2:    $children \leftarrow children(v)$ 
3:    $siblings \leftarrow siblings(v)$ 
4:    $X' \leftarrow X, PG'_S \leftarrow PG_S$ 
5:    $V_{PG'} \leftarrow V_{PG} \setminus v$ 
6:    $E_{PG'} \leftarrow E_{PG} \setminus \{e : v - v', v' \in V_{PG}\}$ 
7:   for all  $v' \in children$  do
8:      $(PG'_S, X') \leftarrow REMOVE(v', PG'_S, X')$ 
9:   end for
10:  for all  $v' \in siblings$  do
11:     $(PG'_S, X') \leftarrow REMOVE(v', PG'_S, X')$ 
12:  end for
13:  for all  $x \in created(v)$  do
14:     $X' \leftarrow X' \cup x$ 
15:  end for
16:  return ( $X', PG'_S$ )
17: end procedure

```

Missing edges in a given precedence graph are created with `calcDeps` by simply checking all possible *new* pairs of nodes v_i and v_j : either v_j depends on v_i (Line 6), v_i depends on v_j (Line 9), or the nodes are independent of each other.

```

1: procedure CALCDeps( $PG_S, PG'_S$ ):  $PG_S^*$ 
2:    $PG_S^* \leftarrow PG'_S$ 
3:   for all  $v_i \in V_{PG^*} \setminus V_{PG}$  do
4:     for all  $v_j \in V_{PG^*}$  do
5:       if  $created(v_i) \cap context(v_j) \neq \emptyset$  then
6:          $E_{PG_S^*} \leftarrow E_{PG_S^*} \cup [e : v_i \rightarrow v_j]$ 
7:       end if
8:       if  $created(v_j) \cap context(v_i) \neq \emptyset$  then
9:          $E_{PG_S^*} \leftarrow E_{PG_S^*} \cup [e : v_j \rightarrow v_i]$ 
10:      end if
11:    end for
12:  end for
13:  return  $PG_S^*$ 
14: end procedure

```

Example 36 (*Applying update to a precedence graph and a set of new elements*). —————

To illustrate how update works, Fig. 5.30 depicts a target graph G_T and its precedence graph PG_T , which is to be updated to include derivations for the `elements` command and `c1`.

The precedence graph is extended to include the single derivation ② for the new elements by invoking `collectDerivations` on Line 3 of `update`. This update is a simple case, as `remove` does not delete any existing derivation from the precedence graph. This is because ② has no *siblings* (derivations that are mutually exclusive) in PG_T , i.e., there exist no other derivations that co-create elements with ②.

Finally, a single edge from ① to ② is created with `calcDep` and the update procedure terminates and returns the precedence graph.

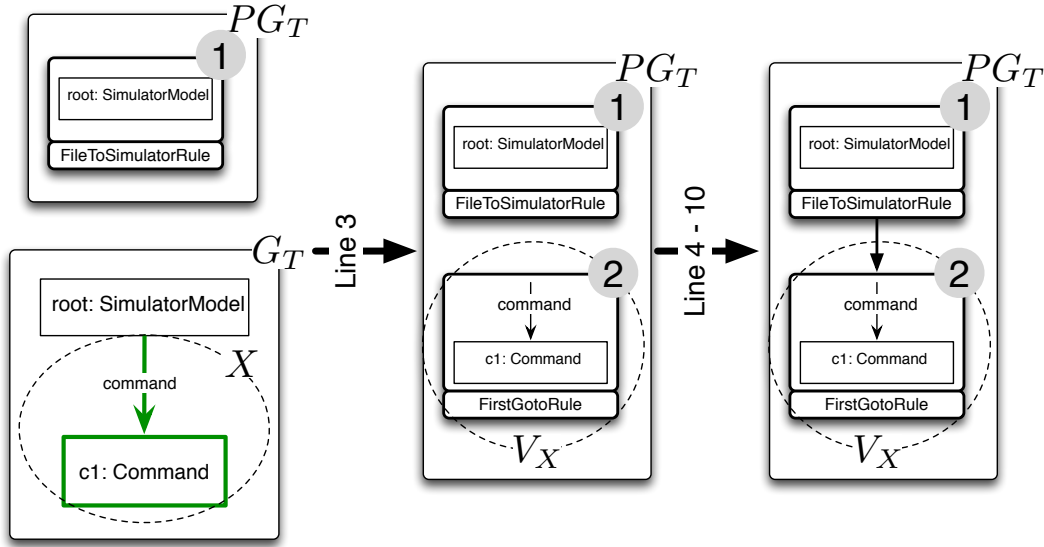


Figure 5.30: A simple update sequence

A more challenging update is depicted in Fig. 5.31. In this case the target graph G_T is extended further with a new Command that is, however, added *at the beginning of the chain*. This is a costly change as the existing derivation for `c1` is now invalid; it would now be blocked by a DEC NAC!

After collecting derivations for all new elements on Line 3, the precedence graph now contains 4 derivations. The existing derivation ② is now a sibling of ③ and is, therefore, removed and not re-collected on Lines 4-9 of `update`.

Dependency edges are created on Line 10 to result in the final target precedence graph, which was already presented and explained in Fig. 5.28.

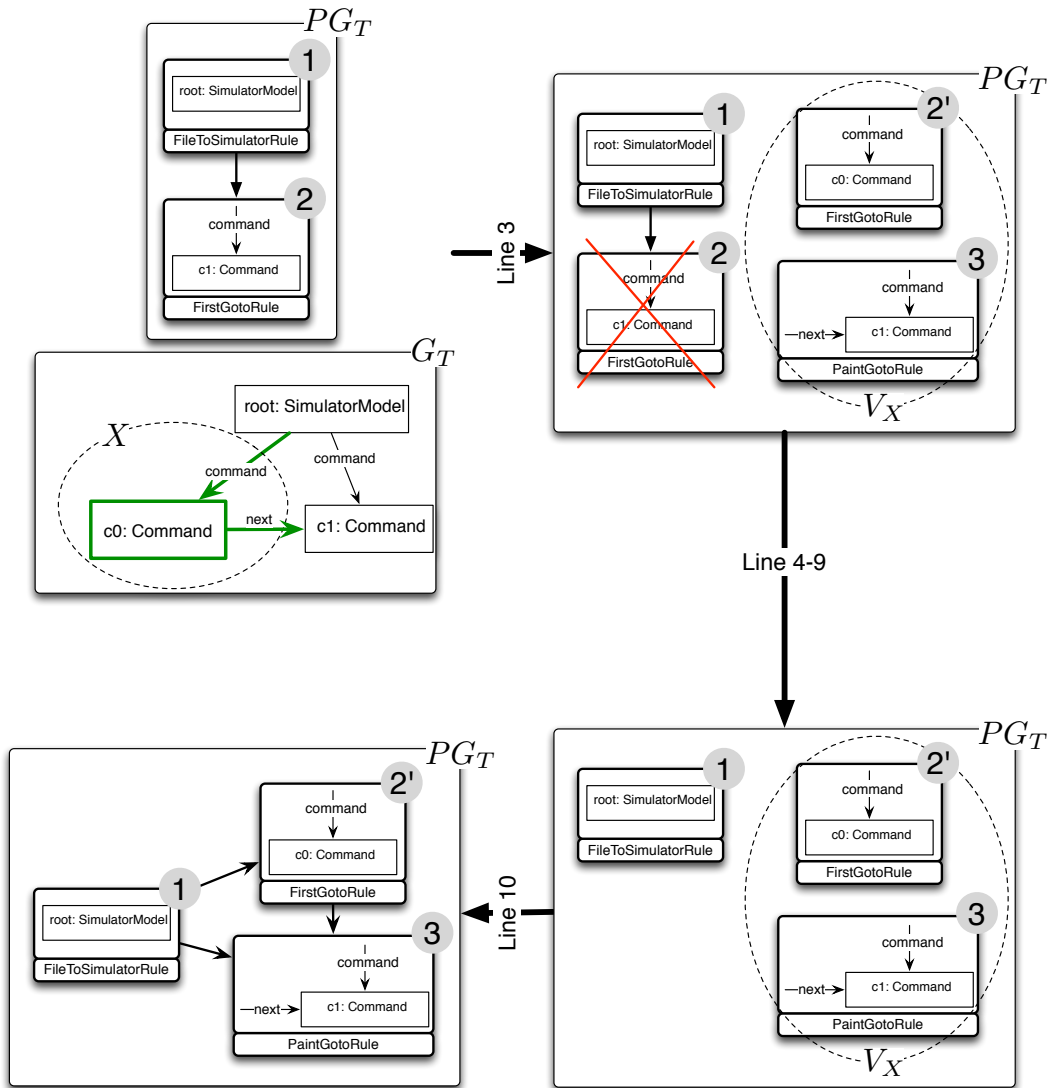


Figure 5.31: An update sequence that invalidates existing derivations

As precedence graphs are to be used to guide the synchronization process, it is imperative to prove that update preserves the properties of a precedence graph, i.e., that the updated precedence graph is correct (invalid derivations must be removed), and complete (it contains all possible derivations and all context dependencies).

The following Lemma 4 shows that invalidation of existing derivations can only be due to the addition of new *dangling edges* as in the previous example. The basic idea of the proof is thus to argue that remove handles these cases by removing all siblings and their transitive dependencies from the precedence graph.

Lemma 4 (*Algorithm 6 produces a precedence graph from a precedence graph*).

Proof.

INDUCTION BEGIN: For $X = G_S = \emptyset$, and $PG_S = \emptyset$ as the precedence graph for $G_S \setminus X$, the resulting precedence graph $(X', PG'_S) = \text{UPDATE}(X, G_S, PG_S, \mathcal{R}_S^-) = \emptyset$ is trivially correct, i.e., a precedence graph for G_S according to Def. 56.

INDUCTION HYPOTHESIS: We assume PG_S is a precedence graph for $G_S \setminus X$.

INDUCTION STEP: To show that $(X', PG'_S) = \text{UPDATE}(X, G_S, PG_S, \mathcal{R}_S^-)$ is a precedence graph for G_S , we have to check the three conditions given in Def. 56:

1. Condition (1) demands that all nodes in PG'_S correspond to derivations with source rules. As the condition holds for PG_S , we only have to argue that (i) all nodes *added* to PG'_S correspond to source rule derivations, and (ii) that all nodes that were already in PG_S still correspond to source rule derivations.

Nodes are only added to PG_S on Lines 3 and 8 of Alg. 6. They are precedence matches newly determined using `COLLECTDERIVATIONS` and thus correspond to their induced source derivations according to Def. 54.

The only possibility of violating Condition (1) from Def. 56 is, therefore, via nodes in PG_S that are no longer valid, i.e., no longer correspond to source rule derivations after `UPDATE` terminates.

As elements are, however, only added and not deleted, induced source derivations can only become invalid if a dangling edge condition is violated due to an added edge.

Without loss of generality, assume that only one such dangling edge condition $N \in \mathcal{N}_{\text{DEC}}$ of a node $v \in V_{PG_S}$ is violated after `UPDATE` terminates. This can only happen if the forbidden dangling edge $e : x - y$ has been added, i.e., $e \in X$. Two cases are possible as $x \in X$ or $x \notin X$, but this distinction does not make a difference.

Of the new precedence matches $\bar{v} = [\bar{s}\bar{m}' : \bar{S}\bar{L} \rightarrow \bar{S}\bar{R}]$ that create e , i.e., $e \in \text{created}(v')$, only those that additionally create y , i.e., $y \in \text{created}(v')$, are

potentially harmful as this might now be the only way to create e . Creating y with an existing $v \in V_{PG_S}$ could, therefore, be blocked by N as e could become dangling (might never be created).

To conclude: only existing nodes $v \in PG_S$ that create y can potentially become invalid by adding $e : y \rightarrow x$. This situation is depicted in Fig. 5.32 for an existing v and a newly added \bar{v} . To avoid such a situation, `UPDATE` determines all new precedence matches V_X for all added elements on Line 3. For each $\bar{v} \in V_X$, `REMOVE` determines with `siblings(v)` all existing nodes $v \in V_{PG_S}$ that create an element created by \bar{v} . This is a superset of all potentially invalidated nodes, which are all removed recursively with all their dependencies from the precedence graph. The set X of elements for which precedence matches must be re-collected might of course be enlarged in the process (cf. Line 14 in procedure `REMOVE`). The dangerous situation depicted in Fig. 5.32 is, therefore, impossible and Condition (1) cannot be violated by `REMOVE`.

2. Condition (2) is guaranteed by `CALCDEPS`, which checks all possible pairs of nodes between which new edges might have to be created.
3. Condition (3) demands that PG'_S is complete in the sense that no derivations are missing, and is guaranteed by the procedure `COLLECTTRANSLATIONS` that loops over all elements to be handled and *all* possible precedence matches. This is invoked on Line 3 and 8 of procedure `UPDATE`, for all newly added elements and all elements for which matches must be newly determined, respectively.

□

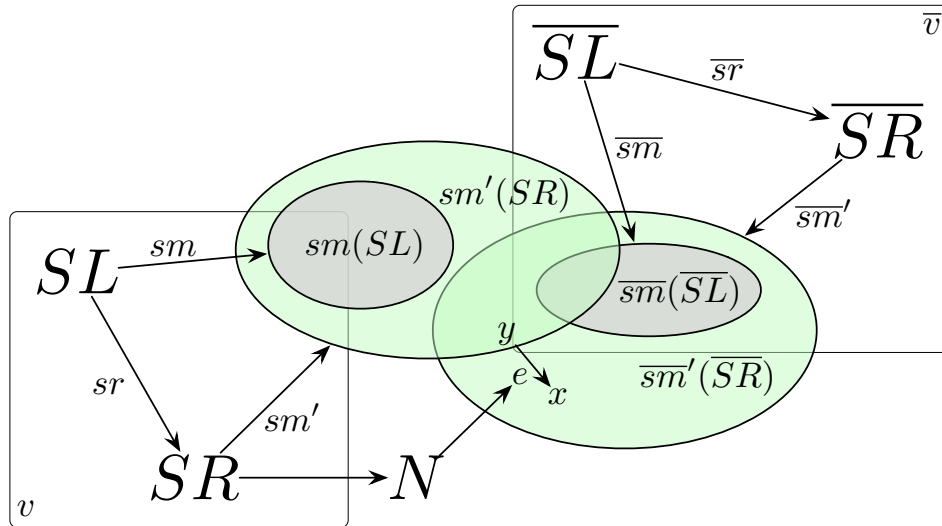


Figure 5.32: Invalidation of precedence match v due to added dangling edge $e : y \rightarrow x$

5.2.4 Using a Translation Protocol to Consistently Revoke Derivations

Based on the established concept of a precedence graph, this section provides algorithms for determining and revoking inconsistent parts of a triple graph, for translating new elements, and, finally, for precedence-based synchronization, which combines and uses all auxiliary algorithms to accomplish respective sub-tasks required for incremental delta propagation.

To handle deletions of elements in a delta, a simple *translation protocol* can be used to determine which TGG rule applications must be revoked. This auxiliary data structure is formalized as follows.

Definition 58 (*Translation Protocol*).

Let $TGG = (TG, \mathcal{R})$ be a triple graph grammar, and $G \in \mathcal{L}(TGG)$ a triple graph.

A *translation protocol* TP for G is a graph $TP = (V_{TP}, E_{TP})$ such that:

1. The set of nodes $V_{TP} = \{v_1, v_2, \dots, v_n\}$ corresponds to a derivation of TGG rules: $\emptyset = G_0 \xrightarrow{r_1} G_1 \xrightarrow{r_2} \dots \xrightarrow{r_n} G_n = G$, such that:
 $\forall i \in \{1, \dots, n\}, v_i = H_{i-1} \xrightarrow{r_i} H_i, H_{i-1} \subset H_i \subseteq G_i$.
2. $\forall (v_i, v_j) \in V_{TP}, \text{created}(v_i) \cap \text{context}(v_j) \neq \emptyset \Leftrightarrow \exists e : v_i \rightarrow v_j \in E_{TP}$,
 where the following relations are defined for $TP, v \in V_{TP}, v = H \xrightarrow{r@m} H'$
 with fulfilled source and target dynamic conditions (i_k, β_k) , a set $V \subseteq V_{TP}$
 of nodes in TP , and a set $X \subseteq G$ of elements:

$$\begin{aligned}
 \text{created}(v) \subseteq G &:= H \setminus H' \\
 \text{created}(V) \subseteq G &:= \bigcup_{v \in V} \text{created}(v) \\
 \text{context}_{DC} &:= [\bigcup_k q_k(C_k)] \setminus \text{created}(v) \\
 \text{context}(v) &:= H \cup \text{context}_{DC} \\
 \text{children}(v) \subseteq V_{TP} &:= \{v' \in V_{TP} \mid \exists e : v \rightarrow v' \in E_{TP}\} \\
 \overline{\text{children}}(v) \subseteq V_{TP} &:= \text{transitive closure over children}(v) \\
 \overline{\text{children}}(V) \subseteq V_{TP} &:= \bigcup_{v \in V} \overline{\text{children}}(v) \\
 \text{creates}(X) \subseteq V_{TP} &:= \{v \in V_{TP} \mid \text{created}(v) \cap X \neq \emptyset\}
 \end{aligned}$$

In contrast to a precedence graph, a translation protocol is not restricted to the source or target domain and tracks both intra- and inter-domain dependencies between elements in a triple graph. Furthermore, a translation protocol contains *actual* dependencies and not potential dependencies such as a precedence graph. Although a translation protocol might seem redundant, after all we have both a source *and* a target precedence graph, it is difficult to prove consistency preservation for deletion handling with a precedence graph, whereas this is almost trivial using a translation protocol.

Finally, the potential dependencies in the precedence graph are not required for handling deletions and can even lead to unnecessary deletions in some cases.

The following algorithm states how a set of elements X can be removed from a consistent triple graph G using a translation protocol. A consistent triple graph G' without any element in X is produced, together with an updated set of “revoked” elements $X' \subseteq X$ that now contains all elements that were removed in addition to those in X to preserve consistency. Lemma 5 ensures that G' is indeed consistent.

Algorithm 7 Revoking Derivations Consistently

Require:

- (a) TGG : operationalizable triple graph grammar
- (b) $G \in \mathcal{L}(\text{TGG})$: consistent triple graph
- (c) $X \subseteq G$: elements to be revoked from translation protocol and G
- (d) TP : translation protocol for G

```

1: procedure REVOKE( $G, X, \text{TP}$ ): ( $G', X', \text{TP}'$ )
2:    $V_X \leftarrow \text{creates}(X)$ 
3:    $V_X \leftarrow \text{children}(V_X)$ 
4:    $X' \leftarrow \text{created}(V_X)$ 
5:    $G' \leftarrow G \setminus X'$ 
6:    $V_{\text{TP}'} \leftarrow V_{\text{TP}} \setminus V_X$ 
7:    $E_{\text{TP}'} \leftarrow E_{\text{TP}} \setminus \{e : v \rightarrow v', v \in V_{\text{TP}}, v' \in V_X\}$ 
8:   return ( $G', X', \text{TP}'$ )
9: end procedure

```

Lemma 5 (*Algorithm 7 preserves consistency*).

If the requirements for Alg. 7 are fulfilled, the following holds:

$$G \in \mathcal{L}(\text{TGG}), (G', X', \text{TP}') = \text{revoke}(G, X, \text{TP}) \Rightarrow G' \in \mathcal{L}(\text{TGG})$$

In addition, TP' is a translation protocol for G' .

Proof. As the translation protocol TP is updated on Lines 6 and 7 by removing all revoked derivations, TP' is a translation protocol for G' .

$G' \notin \mathcal{L}(\text{TGG}) \Rightarrow \exists v \in V_{\text{TP}'}, v = H \xrightarrow{r@m} H'$ is invalid, i.e., r is no longer applicable at m . As TGG is operationalizable (only has source/target NACs), is monotonic creating, and revoke only removes elements, v can only be invalidated due to missing context elements. This is, however, avoided on Line 3 by determining all dependent derivations, whose created elements are then all revoked on Line 5. G' is thus consistent. \square

5.2.5 Restrictions for Precedence-Driven Translation without Backtracking

Now that we can determine and handle the consequences of added and deleted elements in a delta, the next step on the way to precedence-driven synchronization is to be able to *translate* all revoked elements. Forward translation takes a consistent triple graph G , a set of source elements X_S to be translated (these are a mixture of newly added elements and all revoked elements due to additions or deletions), and produces a new triple graph G' with the source graph extended exactly by X_S , and the correspondence and target graphs extended as required to achieve consistency, i.e., $G' \in \mathcal{L}(\text{TGG})$.

In general, the search for the appropriate sequence of rule applications requires backtracking and is, therefore, exponential in runtime with respect to model size. To avoid backtracking during forward translation, we already require source precedence-compatibility, which ensures that the source precedence graph does not induce invalid source derivations that would otherwise be blocked by source NACs. This is, however, insufficient as not all *partial* source derivations induced by the source precedence graph can be extended to a complete source derivation of the source graph. This is partly the reason why backtracking is required to correct local decisions and switch to a different “branch” of rule applications that hopefully produces the complete source graph.

The following definitions formalize *precedence-induced derivations* as all partial derivations that are induced by a precedence graph, and introduce a new property *source local completeness*, which guarantees that every such partial derivation can be extended to a complete derivation of the source graph. Source locally complete TGGs, therefore, do not require backtracking when determining a *source* derivation for a given *source* graph.

Definition 59 (*Precedence-Induced Derivation*).

Let $\text{TGG} = (\text{TG}, \mathcal{R})$ be an operationalizable, precedence-compatible triple graph grammar, \mathcal{R}_S its set of source rules, and $G_S \in \mathcal{L}(\mathcal{R}_S)$, with precedence graph PG_S .

$$\emptyset = G_{S_0} \xrightarrow{\text{sr}_1 @ \text{sm}_1} \dots \xrightarrow{\text{sr}_n @ \text{sm}_n} G_{S_n}, \quad G_{S_0} \subseteq \dots \subseteq G_{S_n} \subseteq G_S$$

is a *precedence-induced derivation* of PG_S if:

- (i) $\exists [v_1, v_2, \dots, v_n] \subseteq V_{\text{PG}_S}$, such that $\forall i \in \{1, \dots, n\}, G_{S_i} \xrightarrow{\text{sr}_i @ \text{sm}_i} G_{S_{i+1}}$,
 v_i is of the form $H_i \xrightarrow{\text{sr}_i @ \text{sm}_i} H_{i+1}$ with $H_i \subseteq G_{S_i}$ and $H_{i+1} \subseteq G_{S_{i+1}}$
- (ii) $\forall j \in \{1, \dots, n\}, \text{context}(v_j) \subseteq \bigcup_{i < j} \text{created}(v_i)$
- (iii) $\forall i \neq j, \text{created}(v_i) \cap \text{created}(v_j) = \emptyset$

The first and second conditions ensure the derivation is induced by (i) choosing nodes in the precedence graph, and (ii) by respecting all dependencies (edges) between nodes if present. The final condition (iii) ensures that every element is only created once, i.e., that an exclusive choice is always made between siblings in the source precedence graph.

Definition 60 (*Source Local Completeness*).

Let $TGG = (TG, \mathcal{R})$ be an operationalizable, precedence-compatible triple graph grammar and \mathcal{R}_S its set of source rules.

\mathcal{R}_S is *locally complete* if:

$\forall G_S \in \mathcal{L}(\mathcal{R}_S)$, where PG_S is the precedence graph for G_S , the following holds:
 $[\emptyset \xRightarrow{*} G'_S \text{ is a precedence-induced derivation of } PG_S] \Rightarrow [\exists G'_S \xRightarrow{*} G_S]$.

TGG is *source locally complete* if \mathcal{R}_S is locally complete.

Target local completeness is defined analogously.

Example 37 (*Illustrating source local completeness with the running example*). —

To provide an intuition for violations of source local completeness, Fig. 5.33 depicts two new Paint rules to replace the previous Paint rules (cf. 5.22). We assume that the TGG developer has decided to demand multiple PAINTs for each Command and allows the integration expert to choose at rule application time between `GotoPaintXYRule` for creating two PAINTs, and `GotoPaintXYZ` for creating three PAINTs. The corresponding target constraints also have to be appropriately adjusted to fit this change. This version of the TGG is not as contrived as it may seem; multiple PAINTs could correspond, e.g., to visualizing the movement in one, two, or all three dimensions, depending on what the user currently prefers. In practice, similar rules are very often specified by (novice) TGG developers.

To explain why these rules are problematic, i.e., why the TGG in this version is *not* source locally complete, a possible target graph G_T and its precedence graph PG_T are depicted in Fig. 5.34.

The target graph G_T consists of a `SimulatorModel` root and a single `Command` `c0` that has three PAINTs `p0`, `p1`, and `p2`. The target precedence graph PG_T has single nodes ① for root and ② for `c0`, and interestingly, *four* siblings for creating the three PAINTs. These four nodes ③, ④, ⑤, and ⑥ represent all possible precedence matches and *potential* source derivations. This demonstrates that precedence graphs do not only contain actual derivations and dependencies but also potential derivations that cannot all be applied to create the given graph.

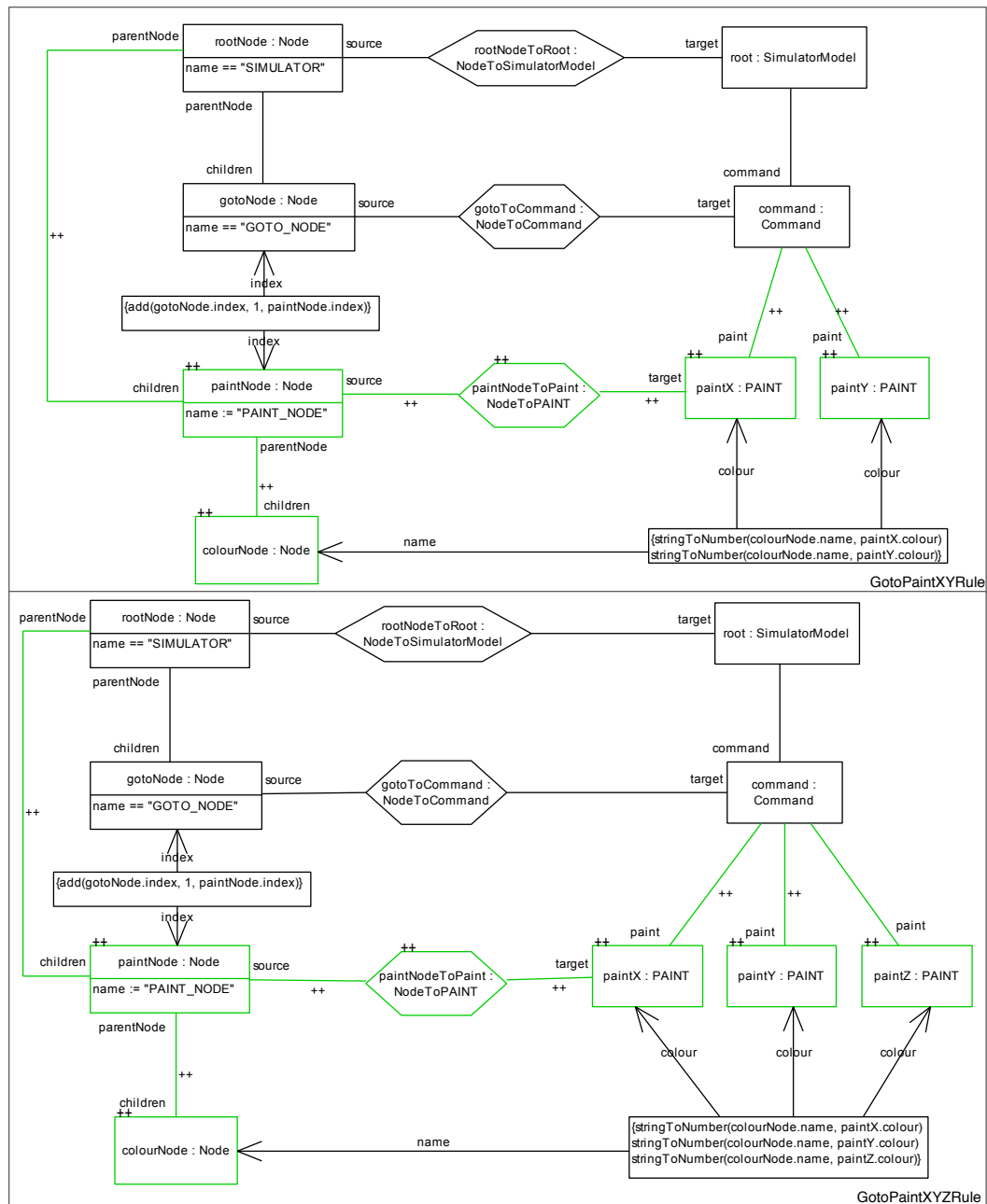


Figure 5.33: Problematic paint rules

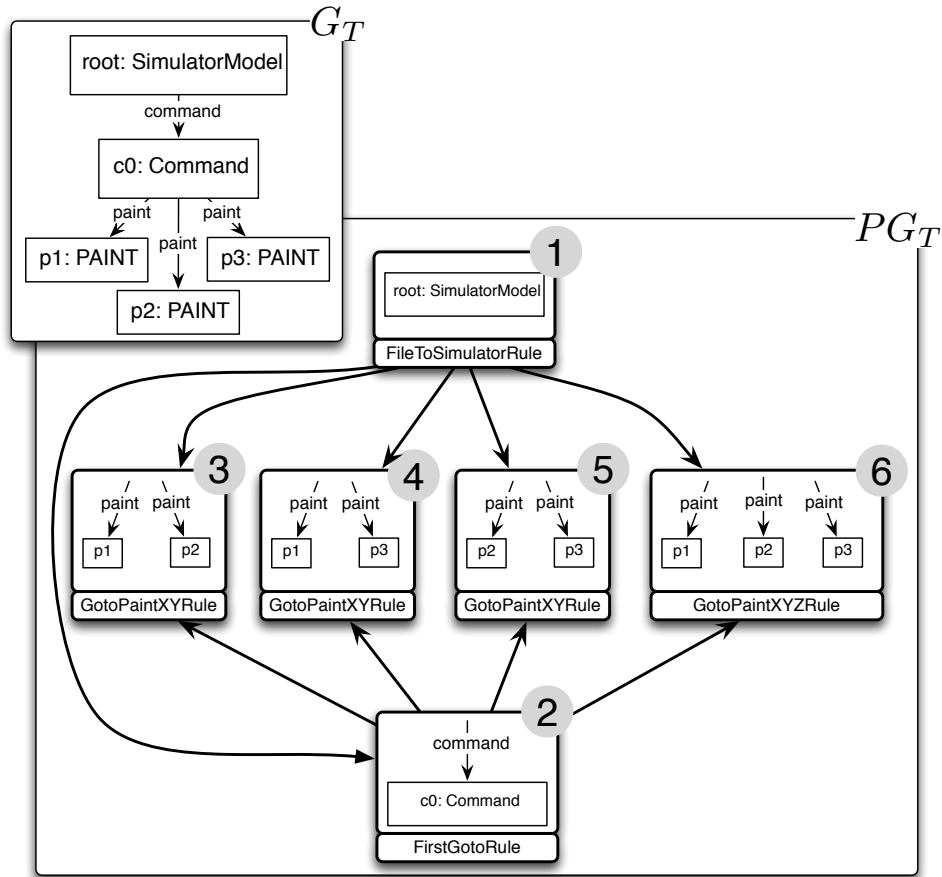


Figure 5.34: Problematic target graph and target precedence graph

According to Def. 59, all the precedence-induced derivations of PG_T are:

- (i) ①: can be extended to ① ② ⑥ (does not violate Def. 60)
- (ii) ① ②: can be extended to ① ② ⑥ (does not violate Def. 60)
- (iii) ① ② ③: cannot be extended to a complete derivation (violates Def. 60)!
- (iv) ① ② ④: cannot be extended to a complete derivation (violates Def. 60)!
- (v) ① ② ⑤: cannot be extended to a complete derivation (violates Def. 60)!
- (vi) ① ② ⑥: is a complete derivation of G_T (does not violate Def. 60)

Due to partial Derivations (iii) – (v), the TGG is not source locally complete. Intuitively, this means that the target precedence graph does not have enough information to appropriately guide the translation process. According to precedences alone, Derivations (iii) – (v) are perfectly fine and the precedence-driven translation algorithm is simply unable to realize that the left-over PAINT, e.g., p_3 in the case of Derivation (iii), can no longer be created by any target rule.

Forward translation with a source locally complete TGG can still hit a dead-end and require backtracking, as there are often multiple source derivations for the same source graph. Not all these source derivations, however, can also be *extended* to complete TGG rule derivations.

As this extension step involves pattern matching over all domains and is quite costly in practice, the following property referred to as *forward local completeness* demands that given a set of direct source derivations, which all create at least one element in common (and are thus pairwise siblings), at least one of the direct source derivations can be extended to a TGG rule derivation.

If this property is fulfilled then an extension of a direct source derivation is never repeated. We know that one of the extensions must be successful due to forward local completeness, and all others will be discarded as only one sibling can be chosen in a precedence-induced derivation.

Definition 61 (*Forward Local Completeness*).

Let $TGG = (TG, \mathcal{R})$ be an operationalizable triple graph grammar, $\mathcal{R}_S, \mathcal{R}_F$ its sets of source and forward rules, respectively, $G, G' \in \mathcal{L}(TGG)$, and PG_S the source precedence graph for G'_S used in the following.

A set V^* of precedence-induced derivations $G \xrightarrow{sr@sm} G'_S \leftarrow G_C \rightarrow G_T, sr \in \mathcal{R}_S$ is *maximal* if $[\bigcap_{v \in V^*} \text{created}(v)] \neq \emptyset$ **and** $\nexists v' = sr'@sm', [\bigcap_{v \in V^* \cup \{v'\}} \text{created}(v)] \neq \emptyset$.

\mathcal{R}_F is *locally complete* if:

V^* is maximal $\Rightarrow \exists sr@sm \in V^*, \exists fr : FL \rightarrow FR \in \mathcal{R}_F,$
 $G'_S \leftarrow G_C \rightarrow G_T \xrightarrow{fr@fm} G'_S \leftarrow G'_C \rightarrow G'_T,$
 such that $sr@sm$ and $fr@fm$ are match consistent.

TGG is *forward locally complete* if \mathcal{R}_F is locally complete.

Backward local completeness is defined analogously.

Definition 62 (*Local Completeness*).

A TGG is *locally complete* if it is source and forward locally complete.

Recall that match consistency of $sr@sm$ and $fr@fm$ implies that there exists a monomorphism $e : SR \rightarrow FL \in \mathcal{M}, sm' = fm \circ e$, and a TGG rule $r = sr *_E fr$ such that $G \xrightarrow{r@m} G'$. This is depicted visually in Fig. 5.35. In this context, e is referred to as a *local extension*, as the source match sm and forward match fm must commute.

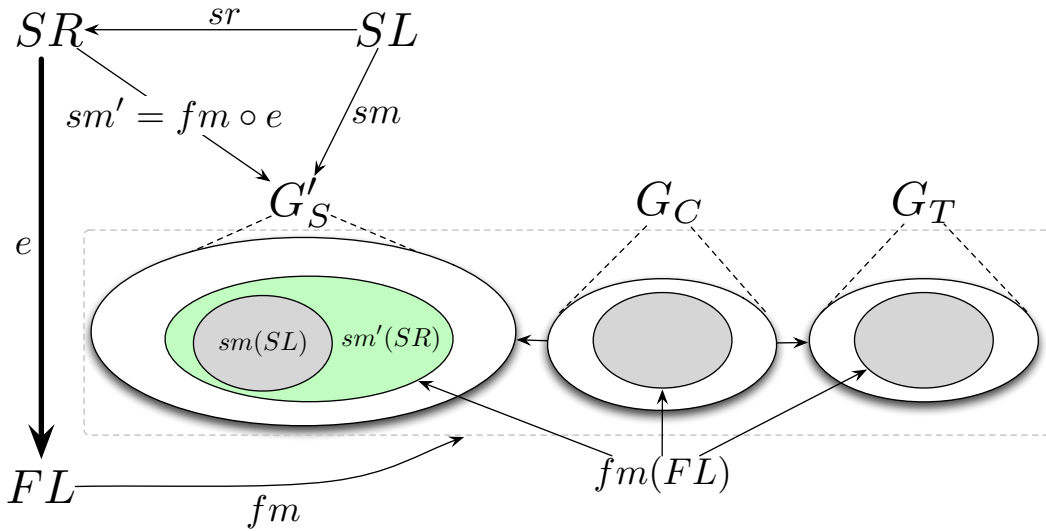


Figure 5.35: Local extension $e : SR \rightarrow FL$ of $sm'(SR)$ to forward match $fm(FL)$

Example 38 (*Understanding forward local completeness*).

Consider the rules `PaintGotoRule::V1` (Fig. 5.21) and `GotoPaintModalRule::V1` (Fig. 5.22) discussed previously. To show why the TGG with these rules is *not* forward locally complete, Fig. 5.36 depicts a consistent triple graph G , a set X_T of target elements to be translated, and the corresponding target precedence graph PG_T that is to be used to guide the translation process.

PG_T has already been updated using the procedure `update` and contains all matches for all new target elements. In addition to nodes ①, ②, and ③ already discussed previously, PG_T now contains nodes that create the PAINT elements p_0 and p_1 . Note that for each PAINT, both target rules for `GotoPaintRule` and `GotoPaintModalRule` (Fig. 5.22) appear to be potential candidates.

Given the triple graph G as a starting point, i.e., ① and ② have already been extended to complete derivations, all possible maximal sets of “ready” target derivations according to Def. 61 are: $V_1^* = \{\textcircled{3}\}$, and $V_2^* = \{\textcircled{4}, \textcircled{5}\}$. If the TGG is forward locally complete, ③ must be extensible to a complete derivation as it is the only target derivation in V_1^* . The only match consistent extension for ③ with the corresponding backward rule fails, however, as it requires the node `paintNode` as source context in the tree (cf. `PaintGotoRule::V1` depicted in Fig. 5.21)! This node is not present, however, because the corresponding PAINT target element p_0 has not yet been translated. In such a case, a precedence-driven translation algorithm could accept that the extension of ③ failed and continue by trying to extend V_2^* . The problem here is that this trial-and-error extension might have to be repeated constantly until the right source derivation is found. To guarantee efficiency, forward local completeness is demanded instead, and `PaintGotoRule::V1`, therefore, constitutes a violation thereof.

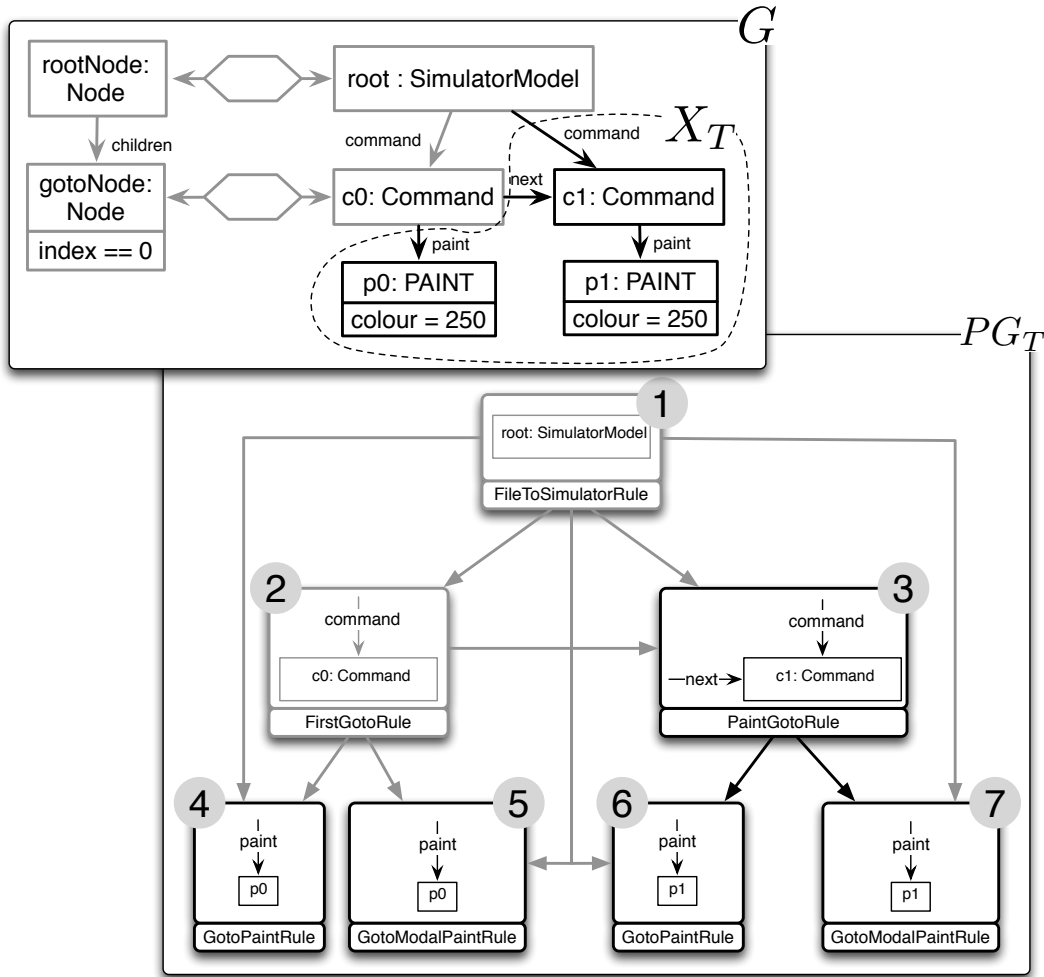


Figure 5.36: Problematic triple graph and target precedence graph

Intuitively, the problem is that the target precedence graph does not have enough context information to appropriately guide the backward translation process. To resolve this violation, additional context information must be added to `PaintGotoRule::V1`, to guarantee that the required source context is also present.

Analogously, a careful consideration of derivations ⑤ and ⑦ in PG_T reveals that the dependency due to the source dynamic condition is also missing. `GotoPaintModalRule` requires the source node `lastNonTrivialColourNode` as context and this dependency is not reflected in the target precedence graph.

The corrected versions of `PaintGotoRule` and `GotoPaintModalRule`, now in Version 2, are depicted in Fig. 5.37. The additional context elements are highlighted with a light grey background. In `PaintGotoRule::V2`, `paint` is added as additional context, whose presence reflects the dependency on `paintNode` in the source domain. In `GotoPaintModalRule::V2`, a new target dynamic condition is added and is implemented to search backwards in the sequence of Commands, starting from `command`, and to retrieve the *first* PAINT element with the *same* colour value as `paint`.

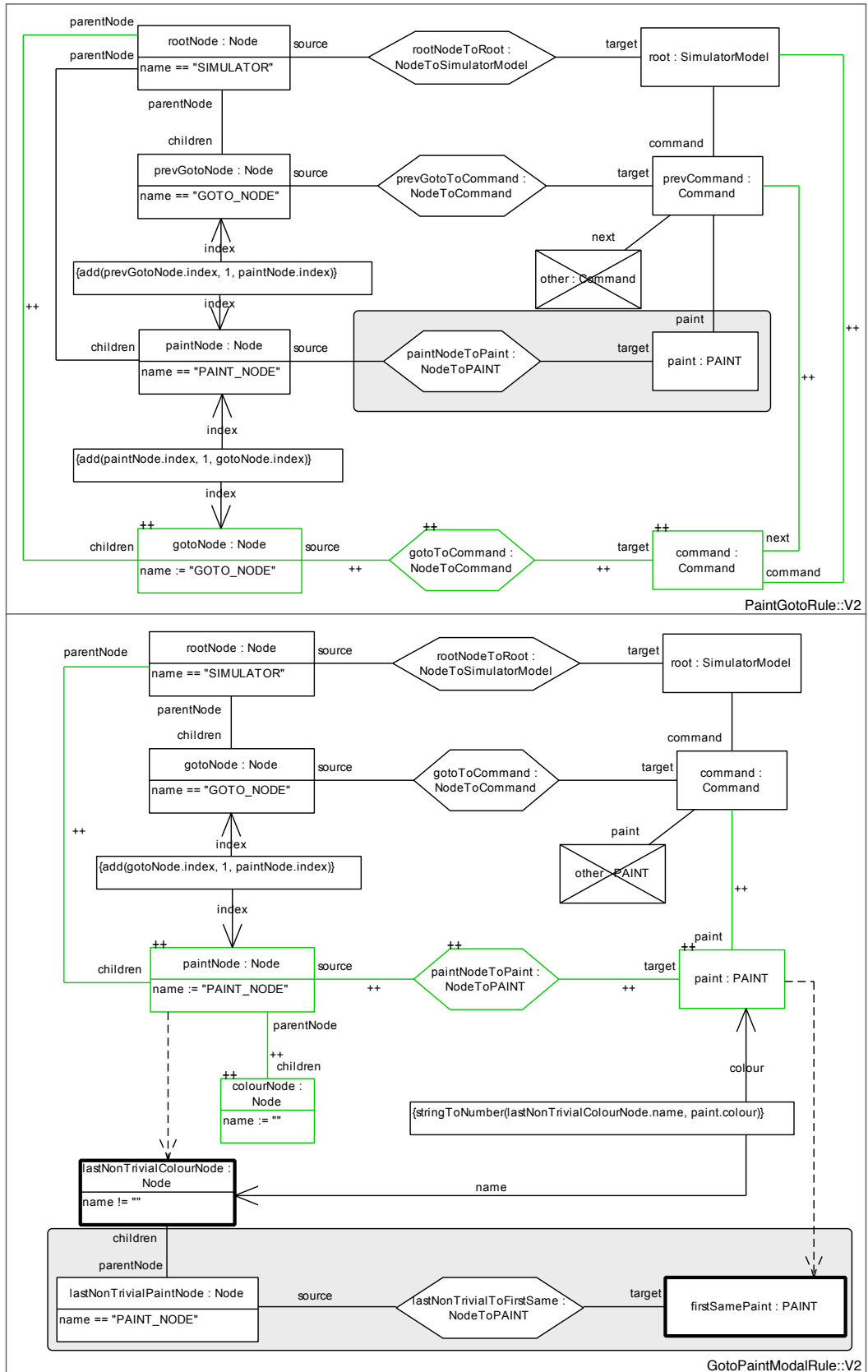


Figure 5.37: Corrected paint rules

To understand this correspondence consider the following sequence of colour node labels in the tree:

$$c0 : 200 \rightarrow c1 : 250 \rightarrow c2 : " " \rightarrow c3 : " " \rightarrow c4 : 200$$

which corresponds to the following sequence of colour values in the model:

$$c0 : 200 \rightarrow c1 : 250 \rightarrow c2 : 250 \rightarrow c3 : 250 \rightarrow c4 : 200$$

In the tree, $c3$ requires the *last non-trivial colour node* which is $c1$, while in the model, $c3$ requires the *first paint node with the same colour value (250)* which is $c1$.

The corresponding target precedence graph PG'_T , now based on the corrected versions of the rules, is depicted in Fig. 5.38. The edge from ④ to ③ now correctly reflects the fact that the paint node created by ④ is indeed required by ③. Analogously, the edge from ④ to ⑦ reflects the dependency of ⑦ on the source node `lastNonTrivialColourNode`. Finally, note that the source derivation ⑤ is now no longer a valid precedence match as it is impossible to satisfy the new target dynamic condition in `GotoPaintModalRule::V2`.

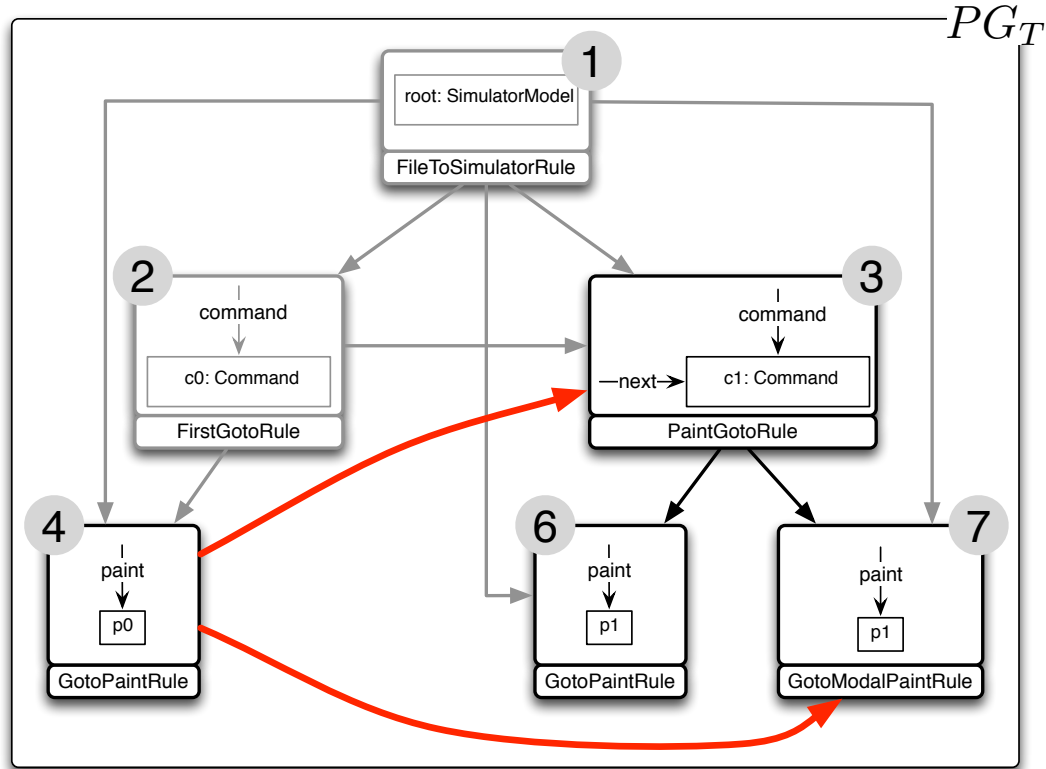


Figure 5.38: Corrected target precedence graph for forward local complete Paint rules

5.2.6 Precedence-Driven Translation

After identifying required properties, namely *source local completeness* and *forward local completeness*, we are now ready to discuss `trans` (Alg. 8), an algorithm for precedence-driven forward translation.

The procedure `trans` takes a set of source elements X_S to be translated, a triple graph G , a source precedence graph PG_S already updated with the elements in X_S , and the set of forward rules to be applied for the forward translation. A translation protocol TP is not required but must be updated in the process for the overall synchronization process.

As a first step, a set `cand` of candidate source derivations is determined on Line 2. This set contains all potential source derivations that can translate at least one element in X_S . The main loop from Line 5 to 21 is repeated until there are no candidates left. After this loop terminates, X_S is expected to be empty (translation was successful) and the process is finalized by calculating any new dependencies in the translation protocol on Line 23 using `calcDeps`. The updated triple graph G' and translation protocol TP' are then returned on Line 24.

In every iteration of the main loop, a set `ready` is determined on Line 6 as all candidates that do not have any incoming edges from nodes in `cand`. For a precedence graph, this means that all context elements have been translated for these elements and exactly these “ready” source derivations can be extended in the next iteration. If all requirements ((a) – (f)) are fulfilled, `ready` is expected to be non-empty.

To continue the translation process, an external component can freely determine a subset `ready*` of `ready`, that is *maximal* according to Def. 61. Although this component is free to pick randomly, it is useful in practice to be able to control this choice, e.g., for reproducible testing and debugging purposes.

As an exclusive choice *must* be made from this set of source derivations in the current iteration (they all create at least one element in common), the attempt to determine an extension performed for each node on Lines 10 – 15 can either succeed or fail, but will *never* be repeated for a derivation in `ready*`. Once again, if all requirements are fulfilled, at least one extension is expected to have succeeded (Line 16).

Finally, one of the extended and now complete derivations v' is chosen and applied to the triple graph on Line 17 by an external component that implements `chooseOneAndApply`. The current iteration is finalized by updating the translation protocol TP accordingly, reducing the sets of candidates by removing all siblings of v' , and of course removing all elements “created” by v' from X_S .

Lemma 6 shows that the conditions required by `TRANS` are sufficient to guarantee that the procedure does not terminate prematurely and that the resulting triple graph is consistent.

Algorithm 8 Precedence-Driven Translation

Require:

- (a) $TGG = (TG, \mathcal{R})$: operationalizable, precedence-compatible, locally complete triple graph grammar
- (b) X_S : source elements to be translated
- (c) $G = G_S \xleftarrow{\sigma_G} G_C \xrightarrow{\tau_G} G_T \in \mathcal{L}(TGG)$: consistent triple graph
- (d) $\exists G' = G'_S \xleftarrow{\sigma_{G'}} G'_C \xrightarrow{\tau_{G'}} G'_T \in \mathcal{L}(TGG)$, where $G'_S = G_S \cup X_S$
- (e) PG_S : precedence graph for $G_S \cup X_S$
- (f) TP : translation protocol for G
- (g) \mathcal{R}_F : forward rules for TGG

```

1: procedure TRANS( $X_S, G, PG_S, TP, \mathcal{R}_F$ ): ( $G', TP'$ )
2:    $cand \leftarrow \{v \in V_{PG_S} \mid created(v) \subseteq X_S \neq \emptyset\}$ 
3:    $G' \leftarrow G, TP' \leftarrow TP$ 
4:    $G'_S \leftarrow G'_S \cup X_S$ 
5:   while  $cand \neq \emptyset$  do
6:      $ready \leftarrow \{v \in cand \mid \nexists e \in E_{PG_S} : v' \rightarrow v, v' \in cand\}$ 
7:     ASSERT( $ready \neq \emptyset$ )
8:      $ready^* \leftarrow \text{CHOOSEONEMAXIMALSET}(ready)$ 
9:      $extended \leftarrow \emptyset$ 
10:    for all  $v = sr@sm \in ready^*, sr : SL \rightarrow SR, fr : FL \rightarrow FR, r = sr *_E fr$  do
11:      if  $\exists fm(FL) \xrightarrow{fr@fm} fm'(FR)$ 
12:        such that  $sr@sm$  and  $fr@fm$  are match consistent then
13:           $extended \leftarrow extended \cup fr@fm$ 
14:        end if
15:    end for
16:    ASSERT( $extended \neq \emptyset$ )
17:    ( $G', v' = sr@sm, fr@fm$ )  $\leftarrow \text{CHOOSEONEANDAPPLY}(extended, G')$ 
18:     $TP' \leftarrow TP' \cup r@m, r = sr *_E fr, m = (sm_S, fm_C, fm_T)$ 
19:     $cand \leftarrow cand \setminus siblings(v')$ 
20:     $X_S \leftarrow X_S \setminus created(v')$ 
21:  end while
22:  ASSERT( $X_S == \emptyset$ )
23:   $TP' \leftarrow \text{CALCDEPS}(TP, TP')$ 
24:  return ( $G', TP'$ )
25: end procedure

```

Lemma 6 (*Algorithm 8 preserves consistency*).

Proof.

As TGG is precedence-compatible, the derivation $v' = sr@sm \in \text{ready}^*$ chosen on Line 17 is valid as it is chosen based on the precedence graph PG_S . As $sr@sm$ and its local extension $fr@fm$ are match consistent, they are E-related and can be composed to $r@m$. Assuming TRANS does not abort for valid input, therefore, it effectively extends the consistent triple graph G in each step by applying a TGG rule $r = sr *_E fr$ on Line 17. This means that the resulting G' is consistent.

We now have to show that Alg. 8 does not abort for valid input. The relevant locations in the algorithm are marked with assertions: (i) on Line 7, (ii) on Line 16 and (iii) on Line 22.

- (i) The assertion on Line 7 is violated when the set ready of nodes without incoming edges in the precedence graph is empty. For a precedence-compatible TGG, this means that the current source rule derivation $\emptyset \xrightarrow{sr_1} \dots \xrightarrow{sr_n} G_{S_n}$ cannot be extended as all candidates in the precedence graph require context that cannot be fulfilled. This cannot happen as the set of source rules \mathcal{R}_S is required to be locally complete.
- (ii) The assertion on Line 16 is violated if no derivation in the determined set of source rule derivations ready^* can be extended to a match consistent forward derivation. In Def. 61, ready^* corresponds to the maximal set V^* of source rule derivations. As \mathcal{R}_F is locally complete, there must exist at least one source derivation that can be extended to a forward derivation. The set extended, therefore, cannot be empty.
- (iii) The final assertion on Line 22 is violated if all relevant derivations in the precedence graph have been eliminated, but not all elements in X_S have been handled by the algorithm. This is similar to the situation in (i), i.e., the current source rule derivation $\emptyset \xrightarrow{sr_1} \dots \xrightarrow{sr_n} G_{S_n}$ cannot be further extended, not because candidate derivations require context that cannot be fulfilled as in (i), but because there are no candidates left in the precedence graph. This situation is also impossible if \mathcal{R}_S is locally complete.

□

5.2.7 Precedence-Driven Synchronization

The main and final algorithm in this chapter is *sync* (Alg. 9), which takes essentially a consistent triple graph $G_S \xleftarrow{\sigma_G} G_C \xrightarrow{\tau_G} G_T$, a consistent source delta $G_S \xleftarrow{\delta_S^-} \Delta_S \xrightarrow{\delta_S^+} G'_S$, and produces a consistent triple graph $G'_S \xleftarrow{\sigma_{G'}} G'_C \xrightarrow{\tau_{G'}} G'_T$, effectively *extending* the source delta Δ_S to a complete delta $\Delta_S \xleftarrow{\delta_S} \Delta_C \xrightarrow{\delta_T} \Delta_T$, which is then *applied* to obtain G' . The synchronization algorithm can be broken down into three major steps:

Step I (Lines 2 – 4) determines all source elements affected by *deletions* and reduces the triple graph G to an intermediate and consistent state G^- without the deleted elements. This is achieved with *revoke* that makes use of the translation protocol TP. Recall that some additional source elements might be revoked in the process, hence the set del^- is returned by *revoke*, together with the updated triple graph and translation protocol TP^- .

Step II (Lines 5 – 9) determines all potential dependencies of *additions* and reduces the triple graph G^- further to an intermediate and consistent state G^+ without the dependencies of additions. This is achieved by first of all removing all revoked source elements from the source precedence graph with the auxiliary procedure *cleanUp*, which does two things: (i) it removes all nodes from the precedence graph that either create or use one of these elements, and (ii) it re-collects all derivations for elements that are meant to remain in the precedence graph, but might now be matched differently than before. The set *toBeTrans* of source elements to be newly translated is determined on Line 6 of *sync* and consists of added elements and elements that were revoked but not deleted. The source precedence graph is updated next with the set on Line 7 using the procedure *update*, which finalizes the source precedence graph and returns a set dep^+ of source elements that had to be removed and re-integrated into the precedence graph due to potential dependencies on added elements. These elements are finally revoked on Line 8, resulting in the intermediate and consistent triple graph G^+ .

In a final Step III (Lines 10 – 14), the set *toBeTrans* is extended with further revoked elements on Line 10, and then translated using *trans* on Line 11, which extends the triple graph G^+ to a final consistent triple graph G' . To ensure that a *backward* synchronization is possible after the algorithm terminates, the target precedence graph is cleaned up (Line 13) and updated (Line 14).

The consistent triple graphs G, G^-, G^+ , and finally G' are depicted visually in Fig. 5.39 (a) showing their connections. Fig. 5.39 (b) represents the graphs computed by the algorithm in the delta structure introduced in Chapter 2.

The steps taken by the algorithm can be seen clearly in Fig. 5.39 (a), starting with the input triple G , all deleted elements are removed from G_S , and an intermediate consistent triple G^- is determined. Note that more source elements might be revoked (deleted and added again) in G_S^- as compared to Δ_S , as they might have to be translated differently. Additions are handled in the next step resulting in a consistent G^+ , which might again contain fewer elements than G^- . The final result is obtained by translating all added elements, including elements that were revoked and added again, to obtain the final consistent triple G' .

Algorithm 9 Forward Synchronization

Require:

- (a) $TGG = (TG, \mathcal{R})$: operationalizable, precedence-compatible, schema-compliant, locally complete
- (b) $G = G_S \xleftarrow{\sigma_G} G_C \xrightarrow{\tau_G} G_T \in \mathcal{L}(TGG)$: a consistent input graph triple
- (c) $\Delta S = G_S \xleftarrow{\delta_S^-} \Delta_S \xrightarrow{\delta_S^+} G'_S$: a consistent input source delta
- (d) PG_S, PG_T : source and target precedence graphs of G_S and G_T , respectively
- (e) TP : the translation protocol for G
- (f) \mathcal{R}_S^- : the set of source rules for TGG
- (g) \mathcal{R}_T^- : the set of target rules for TGG
- (h) \mathcal{R}_F : the set of forward rules for TGG

```

1: procedure SYNC( $G, \Delta S, PG_S, PG_T, TP, \mathcal{R}_S^-, \mathcal{R}_F$ ): ( $G', PG_S^+, PG_T^+, TP'$ )
2:    $del_S \leftarrow G_S \setminus \Delta_S, add_S \leftarrow G'_S \setminus \Delta_S$ 
3:    $(G^-, del^-, TP^-) \leftarrow \text{REVOKE}(G, del_S, TP)$ 
4:   ASSERT( $G^- \in \mathcal{L}(TGG)$ )
5:    $PG_S^- \leftarrow \text{CLEANUP}(del_S^-, PG_S, G_S^-, \mathcal{R}_S^-)$ 
6:    $toBeTrans \leftarrow add_S \cup [del_S^- \setminus del_S]$ 
7:    $(dep_S, PG_S^+) \leftarrow \text{UPDATE}(toBeTrans, G_S^-, PG_S^-, \mathcal{R}_S^-)$ 
8:    $(G^+, dep^+, TP^+) \leftarrow \text{REVOKE}(G^-, dep_S, TP^-)$ 
9:   ASSERT( $G^+ \in \mathcal{L}(TGG)$ )
10:   $toBeTrans \leftarrow toBeTrans \cup dep_S^+$ 
11:   $(G', TP') \leftarrow \text{TRANS}(toBeTrans, G^+, PG_S^+, TP^+, \mathcal{R}_F)$ 
12:  ASSERT( $G' \in \mathcal{L}(TGG)$ )
13:   $PG_T^- \leftarrow \text{CLEANUP}(G_T \setminus G_T^+, PG_T)$ 
14:   $(X', PG_T^+) \leftarrow \text{UPDATE}(G_T^+ \setminus G_T^+, G_T^+, PG_T^-, \mathcal{R}_T^-)$ 
15:  return ( $G', PG_S^+, PG_T^+, TP'$ )
16: end procedure

```

```

1: procedure CLEANUP( $X, PG_S, G_S, \mathcal{R}_S$ ):  $PG'_S$ 
2:    $PG'_S \leftarrow PG_S$ 
3:    $X' \leftarrow \emptyset$ 
4:   for all  $\{v \in V_{PG_S} \mid [created(v) \cup context(v)] \cap X \neq \emptyset\}$  do
5:      $(PG', X') \leftarrow \text{REMOVE}(v, X', PG'_S)$ 
6:   end for
7:    $V_{X'} \leftarrow \text{COLLECTDERIVATIONS}(X' \setminus X, G_S, \mathcal{R}_S)$ 
8:    $V_{PG'_S} \leftarrow V_{PG'_S} \cup V_{X'}$ 
9:    $PG'_S \leftarrow \text{CALCDEPS}(PG_S, PG'_S)$ 
10:  return  $PG'_S$ 
11: end procedure

```

Fig. 5.39 (b) shows that this process can be viewed as indirectly determining the delta $\Delta = \Delta_S \leftarrow G_C^+ \rightarrow G_T^+$ representing all elements that can be retained during the synchronization. Most interesting is how the morphism $G_C^+ \rightarrow \Delta_S$ is derived via the intermediate steps of the algorithm.

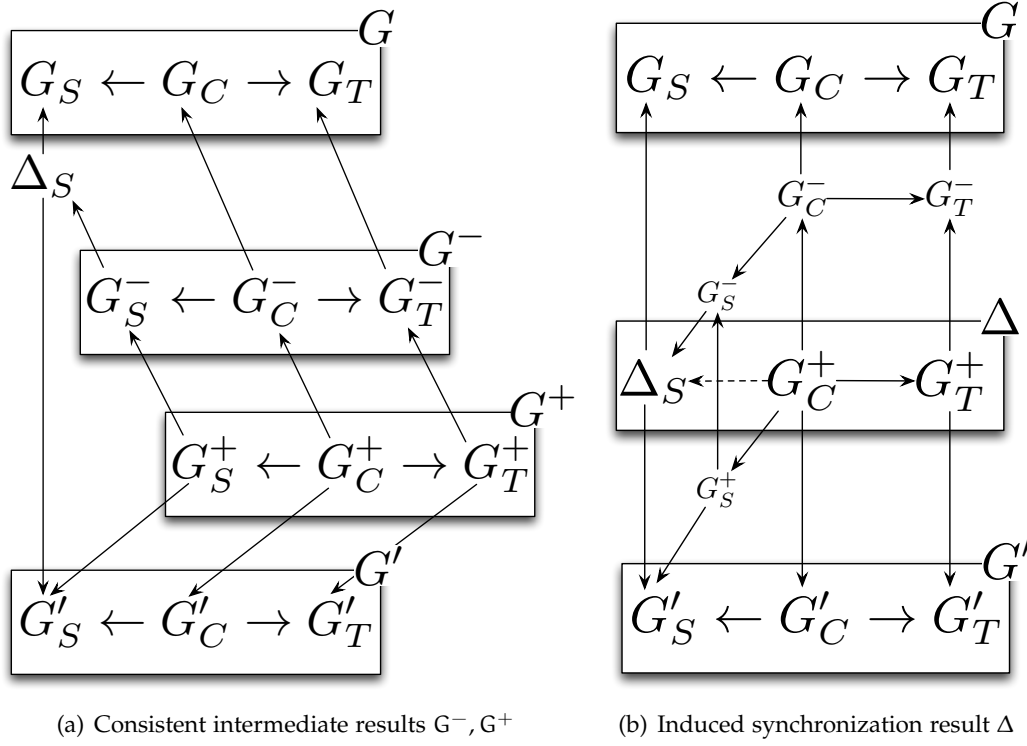


Figure 5.39: Intermediate steps and corresponding states during forward synchronization

Example 39 (Applying sync to propagate a source delta).

To illustrate the complete synchronization process with a concrete example, Fig. 5.40 depicts an input triple graph G , together with a source delta Δ_S , which is superimposed on G using colours and a ++/- - markup such as for rules. Note that in this and following figures related to the example, some details are omitted intentionally to improve readability, e.g., as all edges in the source model are of type children this information is omitted.

The source node $cN0$ and its incident edge in G are to be deleted (red and annotated with - -), while a new source node $cN0'$ and its incident edge are to be created (green and annotated with ++). The source delta thus represents a replacement of the colour node of value 250, with a new colour node of value 300. This is an interesting delta involving dynamic conditions as $cN1$ depends on $cN0$; its colour value was omitted in the CLS file and must be treated as a *modal* paint command. In the target model, which is always explicit, both elements $p0$ and $p1$ consequently have the same colour value 250.

In Step I of sync, the translation protocol for G is used to revoke all deleted elements and their dependencies from G resulting in the triple graph G^- depicted in Fig. 5.41.

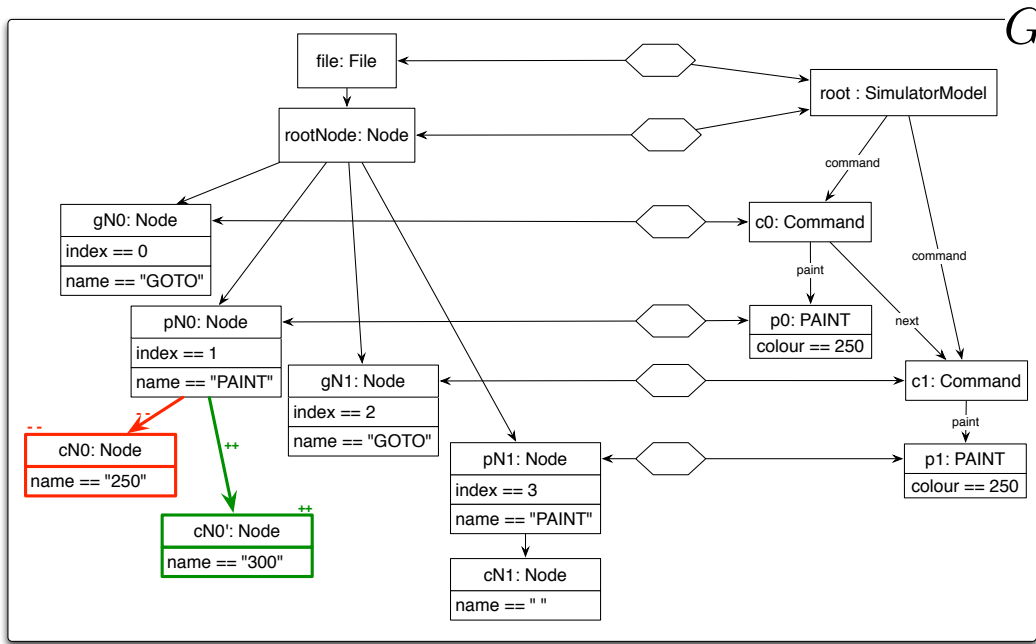


Figure 5.40: Initial triple graph and source delta

The translation protocol TP for G is depicted in Fig. 5.42. The initial triple graph was constructed by applying the five TGG rule derivations in the order: ①②③④⑤. All context dependencies are represented in the translation protocol as edges. Note the edge between ③ and ⑤, which is due to the dynamic conditions of *GotoModalPaintRule* (Fig. 5.37). As $cN0$ is created by derivation ③ in TP, derivations ③, ④, and ⑤ are all revoked.

In terms of sync del_S consists of $cN0$ and its incident edge, whereas del_S^- also includes all source elements that had to be revoked as a consequence, i.e., $pN0$, $gN1$, $pN1$, $cN1$, and all incident edges.

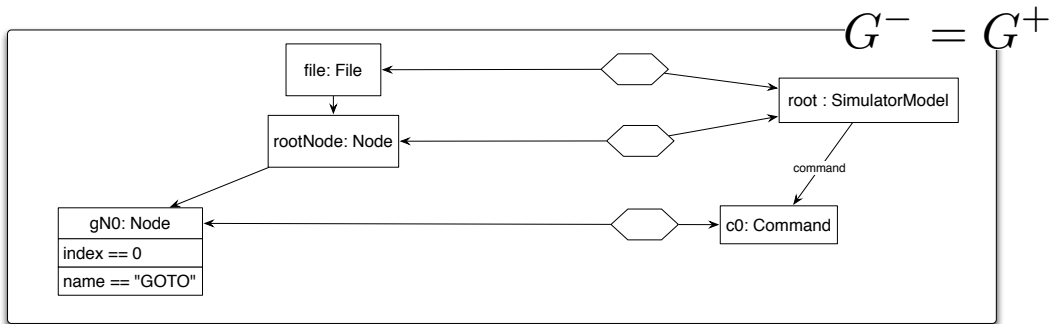


Figure 5.41: Triple graph after Step I

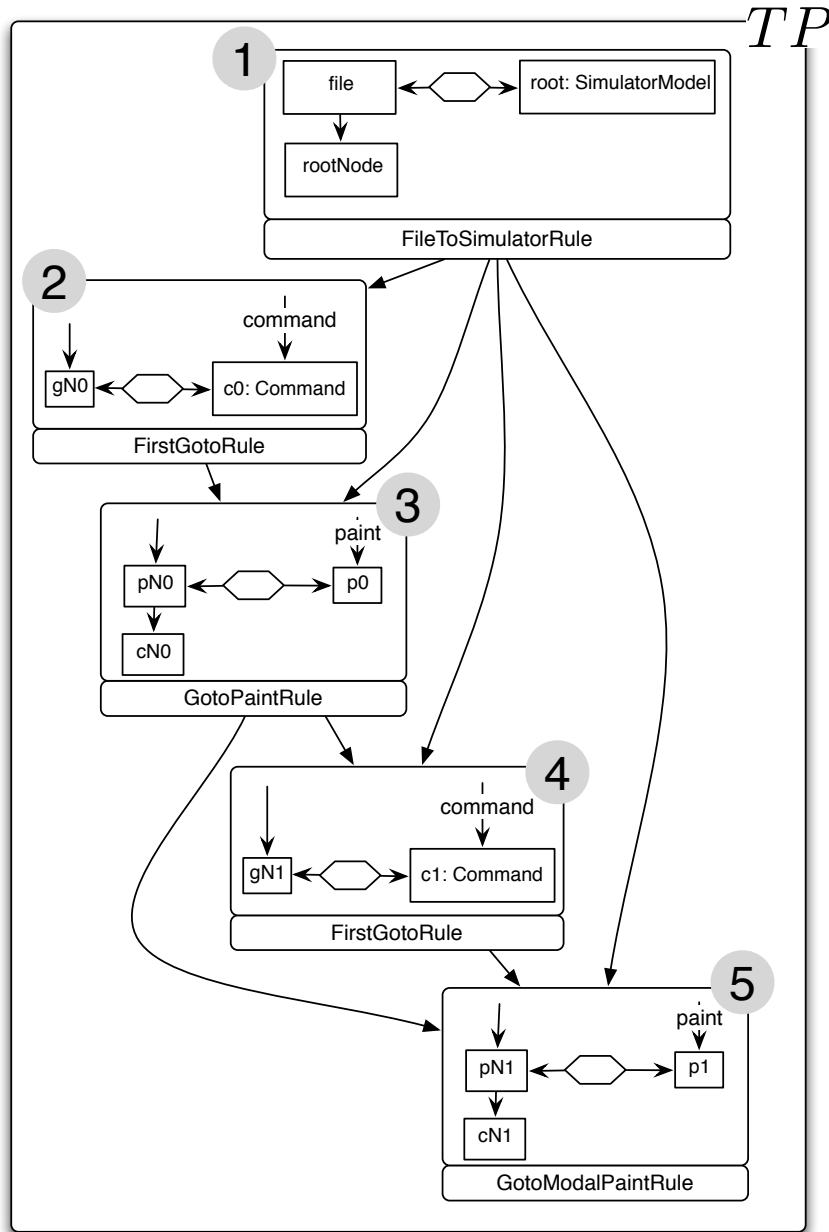


Figure 5.42: Translation protocol for synchronization

The source precedence graph PG_S and target precedence graph PG_T are depicted in Fig. 5.43. Note that PG_T includes an additional target rule derivation ⑥ representing the fact that PAINT target elements can always be translated either modally or not.

This freedom of choice is not present in PG_S as the value of the colour node (a string representing a number or " ") determines exactly which Paint rule must be used. In contrast to the translation protocol, these potential dependencies are important to handle the consequences of added elements correctly.

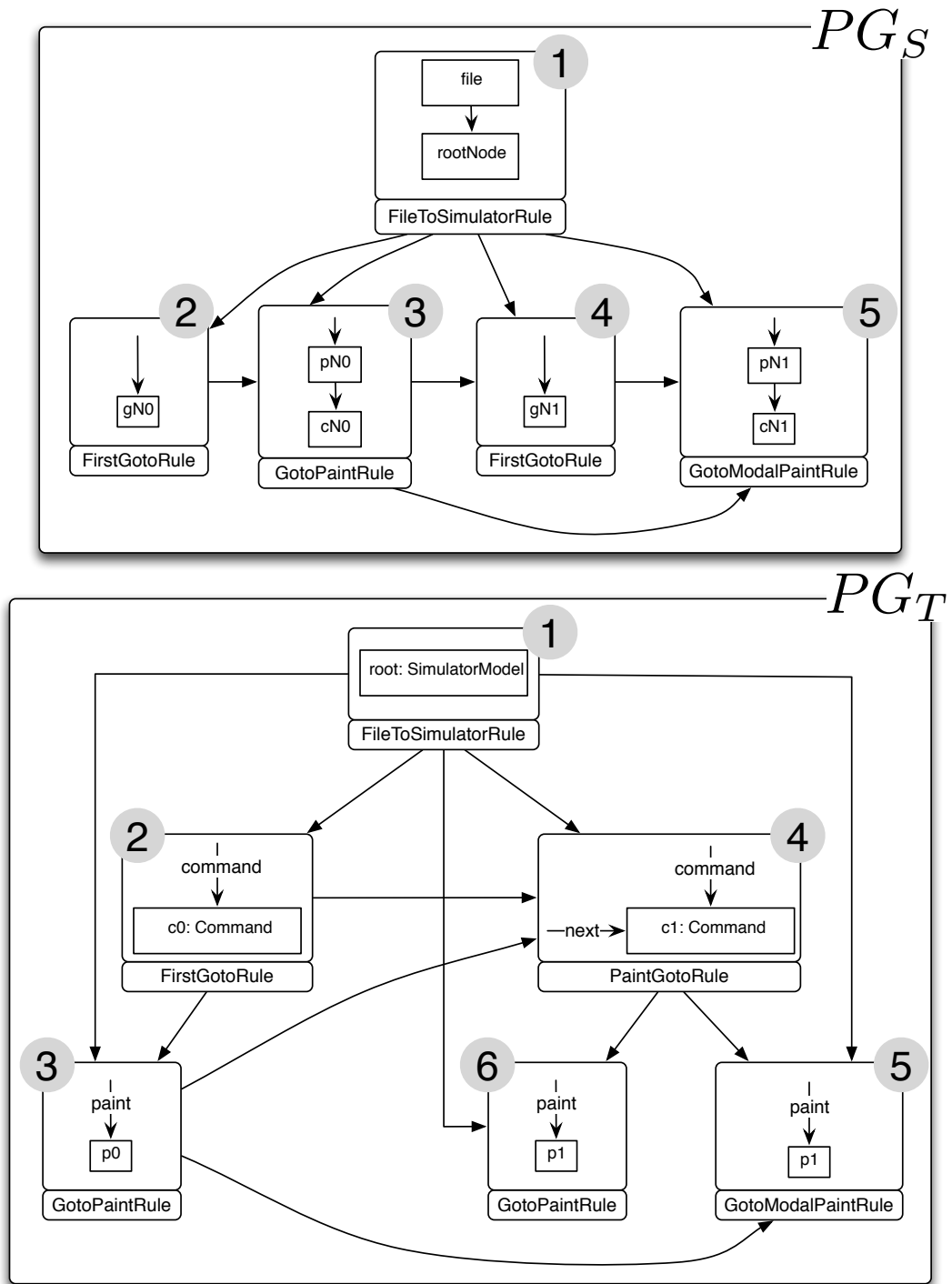


Figure 5.43: Precedence graphs for synchronization

To start Step II, the source precedence graph must be cleaned up by removing all source derivations for revoked elements. The resulting source precedence graph PG_S^- after invoking `cleanUp` is depicted in Fig. 5.44, together with the corresponding translation protocol TP^- .

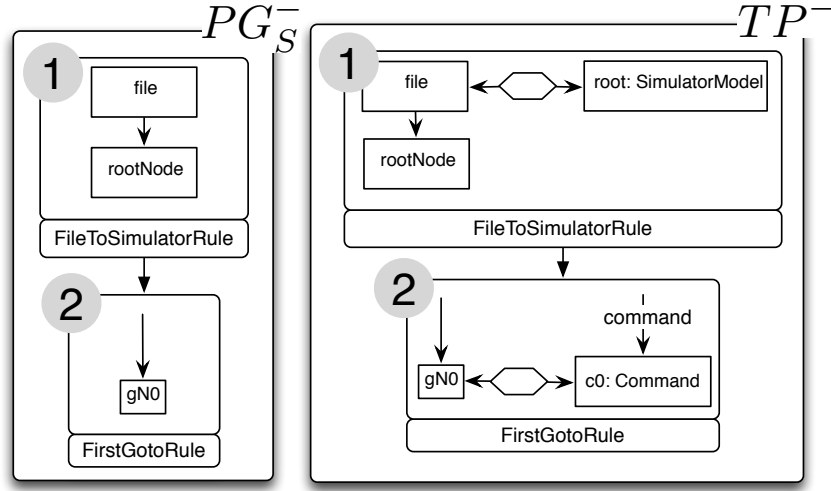


Figure 5.44: Translation protocol and source precedence graph after Step I

PG_S^- is now updated to cover once again all revoked (but not deleted!) source elements as well as the newly added elements (in this case $cN0'$ and its incident edge). The finalized source precedence graph PG_S^+ produced by update is depicted in Fig. 5.45. All new nodes $(3')$, $(4')$, and $(5')$ are emphasized with a bold border. Note the added elements $cN0'$ and its incident edge in $(3')$. As the added elements do not introduce any potentially dangling edges, no further elements must be additionally revoked. This means that $G^- = G^+$ as depicted and discussed previously in Fig. 5.41.

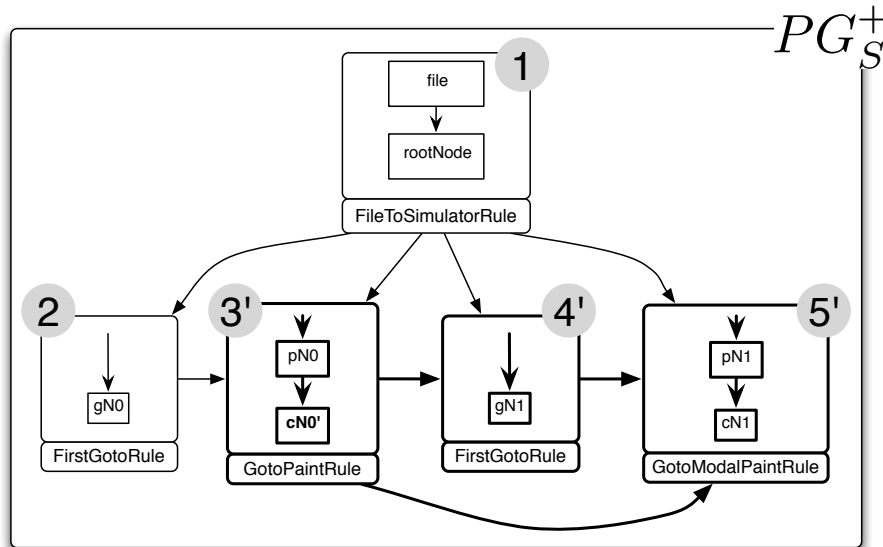


Figure 5.45: Finalized source precedence graph

In the final step, G^+ is translated to G' by invoking trans with PG_S^+ and the set of source elements to be translated (all elements created by $\textcircled{3'}$, $\textcircled{4'}$, and $\textcircled{5'}$). In this case, the derivations can only be applied in the order $\textcircled{3'}\textcircled{4'}\textcircled{5'}$ without any freedom of choice. The resulting triple graph G' is depicted in Fig. 5.46. The newly created elements are emphasized with a bold frame. The PAINT elements in the target model have been updated to have a colour value of 300.

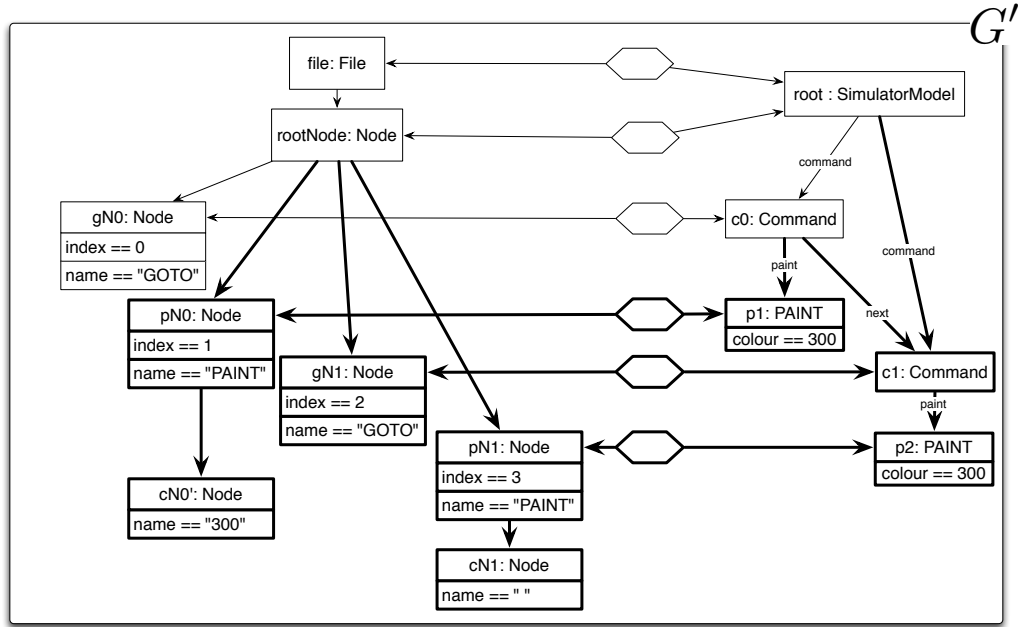


Figure 5.46: Synchronized triple graph

5.2.8 Formal Properties of Derived TGG-Based Synchronizers

To conclude this chapter on the *derivation* of incremental synchronizers from TGG specifications, this section discusses the formal properties introduced in Chapter 2, Def. 20. As we have already discussed consistency preservation of the main algorithms, correctness and completeness are fairly straightforward to show:

Theorem 7 (*Synchronization with Alg. 9 is Correct*).

Proof.

$G \in \mathcal{L}(\text{TGG}) \xrightarrow{\text{Lemma 5}} G^- \in \mathcal{L}(\text{TGG})$ (Line 4) $\xrightarrow{\text{Lemma 5}} G^+ \in \mathcal{L}(\text{TGG})$ (Line 9) $\xrightarrow{\text{Lemma 6}} G' \in \mathcal{L}(\text{TGG})$ (Line 12). \square

Theorem 8 (*Synchronization with Alg. 9 is Complete*).

Proof. ΔS is consistent $\xrightarrow{\text{Lemma 6}} \text{Sync}$ does not abort. \square

Remark 2 (*On the Efficiency of Alg. 9*).

Providing a formal proof for the efficiency of `sync` according to Def. 20 is out-of-scope for this thesis. It is also questionable how useful such a detailed analysis is in practice, where many other technical factors play an important role. The focus in this thesis is on correctness and completeness for the extensions made to the synchronization algorithm of [70]. The interested reader is, therefore, referred to [70] for a detailed complexity analysis.

The following provides an informal discussion of the most important points concerning the efficiency of `sync`, required to argue under what conditions (i) `sync` is polynomial and not exponential in model size, and (ii) `sync` scales with the dependencies of deletions and additions and not with model size:

- If all auxiliary data structures (translation protocol and precedence graphs) are implemented reasonably efficiently, the core task performed by `sync` should be pattern matching. We can thus concentrate for a complexity analysis on `cleanUp`, `update`, and `trans`, the only algorithms that perform pattern matching using `collectDerivations` and `chooseOneAndApply`.

This assumes that all attribute and dynamic conditions are relatively efficient and do not become bottlenecks of the synchronization. As these conditions are, however, designed to be black-box extensions that can be implemented in e.g., Java, this cannot be enforced and must be demanded as part of the contract between the integration expert and the TGG tool developer.

- The effort for performing pattern matching is in the worst case $O(n^k)$ where n is the number of elements from which a match can be constructed, and k is the maximal pattern size of all rules. For `sync` to be efficient according to Def. 20, we must demand here that the degree of all context nodes be bounded and not grow with model size. This restriction forbids global “container nodes”, which are connected to every element in a model. If such elements are forbidden, then n can be taken as the set of all elements affected by all dependencies of deletions and additions in both precedence graphs.
- `cleanUp` and `update` are then at most $O(n^k)$ as pattern matching is only required for a subset of these elements.
- `trans` is also in $O(n^k)$ due to three main properties of the algorithm:
 - (i) source precedence matches are *never* extended twice, extension either fails or succeeds and is never repeated (cf. Line 12 of `trans`), (ii) extension of `ready*` never fails, i.e., at least one source match can always be extended, and (iii) all elements can be translated in this manner. To summarize, backtracking (leading to worst-case exponential runtime) is never required to correct mistakes.

Remark 3 (*On the Incrementality of Synchronization with Alg. 9*).

It is currently impossible to provide a formal proof of incrementality as defined in Def. 20. From a practical point of view, sync is, however, at least as incremental as a naïve batch algorithm that just deletes and re-creates all elements, and is in practice much better (cf. evaluation).

The “problem” here is that precedence-graphs track *potential* dependencies. sync is, therefore, not always as incremental as possible, even with the simple metric for comparing deltas introduced in Def. 20. In other words, incrementality is currently compromised to guarantee correctness.

An optimization idea that improves incrementality without compromising correctness is to restrict COLLECTDERIVATIONS to collecting only direct derivations that *create* one of the elements to be added to the precedence graph, i.e., Line 5 is simplified to just $x \in \text{created}(sm')$. This optimization means that new matches that use a newly added element as context but do not create *any of the newly added elements* will be missing from the updated precedence graph. This is, however, not problematic as all the elements created by such missing matches have already been consistently translated and will not be revoked. The matches could have been in the precedence graph and simply not have been chosen (as other siblings were chosen instead). Collecting these matches can thus be skipped, continuing the synchronization process with a precedence graph that is only “virtually” complete. The process remains correct and complete, the only consequence being that earlier decisions between siblings are never revoked, even when new possibilities arise later in the synchronization process. This freedom of choice is traded-off for increased incrementality (less is revoked) and improved efficiency (fewer matches are collected).

An extension of this idea would be to ensure in UPDATE that a match m really contains an edge that can violate Dangling Edge Check (DEC), *before* invoking REMOVE(f) or the match m . If no such edge exists, then such matches do *not* have to be removed for correctness.

Future work includes providing an even better (fine granular) update algorithm to determine dependencies in the precedence graphs, and a consistency check to ensure that elements are only revoked if they cannot be reused later in the synchronization process. Both ideas, however, make proving correctness challenging and might have an adverse effect on efficiency in practice (in many cases it is probably faster to just revoke and re-translate elements without a costly analysis).

CONSTRUCTION TECHNIQUES AND STATIC ANALYSES

The goal of this chapter, based on [6, 8] by Anjorin et al., is to leverage existing theory from the mature field of algebraic graph transformations for a static analysis of all properties required for precedence-driven synchronization.

Section 6.1 presents a construction technique for NACs used to produce *schema-compliant, precedence-compatible* TGGs given negative source and target constraints, and a TGG without application conditions. Based on the well-known Critical Pair Analysis (CPA), Sect. 6.2 provides a sufficient condition for *source local completeness*. Finally, Sect. 6.3 states conditions that ensure *forward local completeness*.

By establishing the theory required for a static analysis of all required properties identified in the previous Chap. 5, this chapter provides the second part of CONTRIBUTION III addressing CHALLENGE III of this thesis as identified in Sect. 1.3:

Provide a constructive, coherent formal underpinning of all concepts and language constructs, which can serve as a guide for a corresponding implementation and static analysis.

6.1 PRECEDENCE-COMPATIBILITY AND SCHEMA-COMPLIANCE

The following fact presents a well-known construction technique, discussed in detail in [28], with which sufficient and necessary application conditions can be produced to guarantee that a given set of constraints is not violated by any derivation of a given graph grammar. The construction is simplified appropriately for the case of TGGs (non-deleting rules) and conditions as introduced in Def. 9.

Fact 11 (*Construction of Application Conditions from Constraints*).

Given a graph grammar $GG = (TG, \mathcal{R})$ without application conditions, and a set \mathcal{C}_{TG} of constraints over TG .

There is a construction \mathcal{A} producing a set \mathcal{C}_{GG} of application conditions:

$$\mathcal{C}_{GG} = \mathcal{A}(GG, \mathcal{C}_{TG}), \text{ such that } \mathcal{L}(GG, \mathcal{C}_{GG}) \subseteq \mathcal{L}(TG, \mathcal{C}_{TG})$$

The constructed application conditions \mathcal{C}_{GG} are sufficient and necessary.

Proof. This is a special case of Thm. 7.23 in [28]. The interested reader is, therefore, referred to [28] for a detailed proof in a more general setting.

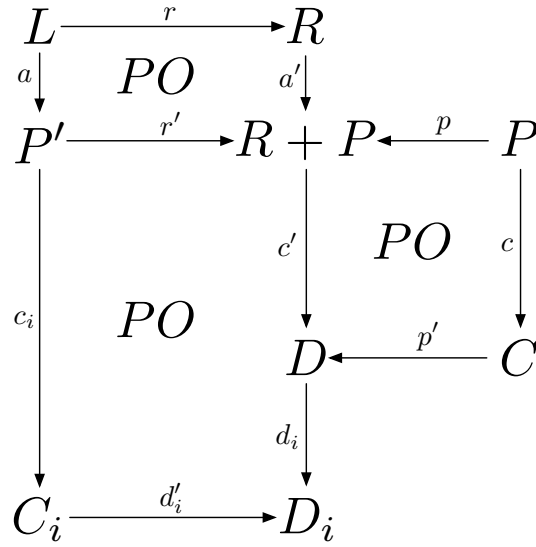


Figure 6.1: Construction of application conditions from constraints

Figure 6.1 depicts the main steps of the construction for a given rule $r : L \rightarrow R$ and constraint $c : P \rightarrow C$, namely:

1. Constructing all possible gluings $R + P$ of the right-hand side R of the rule r and the premise P of each constraint c ,
2. Producing for each gluing $R + P$ a postcondition $(a', \forall d_i \circ c')$ via a pushout, where D_i represents all possible further gluings of elements in D , and finally
3. Constructing a precondition $(a, \forall c_i)$ from the postcondition $(a', \forall d_i \circ c')$ by reversing the application of the rule (determining pushout complements P' and C_i). If this is not possible (i.e., a pushout complement does not exist) then the postcondition does not result in an equivalent precondition. In this case, rule application can never violate the postcondition so no precondition is required.

□

This construction technique can be applied to a TGG without application conditions and a set of source and target negative constraints, which must not be violated. As the construction produces sufficient *and* necessary application conditions (in this case source and target NACs) to prevent constraint violations, the TGG together with the constructed NACs is schema-compliant (NACs are sufficient) and precedence-compatible (NACs are necessary). In practice, therefore, the error-prone manual specification of NACs can be completely replaced using this construction. The following corollary formalizes this idea.

Corollary 1 (*Construction of Schema-Compliant and Precedence-Compatible TGGs*).

Let $TGG = (TG, \mathcal{R})$ be a triple graph grammar without application conditions, and $\mathcal{C} = \mathcal{C}_{TG_S} \cup \mathcal{C}_{TG_T}$ a set of negative constraints either over TG_S or TG_T , respectively.

TGG , together with application conditions $\mathcal{C}_{TGG} = \mathcal{A}(TGG, \mathcal{C})$ constructed according to Fact 11, is *operationalizable* (Def. 46), *precedence-compatible* (Def. 52), and *schema-compliant* (Def. 53).

Proof. The construction in Fact 11 is applicable to TGGs as shown in [6].

OPERATIONALIZABILITY: The construction for triple graphs is defined component-wise [6] so source negative constraints are transformed to source NACs (analogously for target). According to Def. 46, TGG together with \mathcal{C}_{TGG} is thus operationalizable.

PRECEDENCE-COMPATIBILITY: Fact 11 guarantees that the constructed NACs are necessary $\Rightarrow \forall G_S \in \mathcal{L}(\mathcal{R}_S) \setminus \mathcal{L}(\mathcal{R}_S, \mathcal{C}_{TGG_S}), G_S \notin \mathcal{L}(TG_S, \mathcal{C}_{TGG_S}) \xrightarrow{\text{Def. 52}} TGG$ together with \mathcal{C}_{TGG} is source precedence-compatible. Analogously, (TGG, \mathcal{C}_{TGG}) is *target* precedence-compatible and, therefore, precedence-compatible.

SCHEMA-COMPLIANCE: From Fact 11, it follows that:

$\mathcal{L}(\mathcal{R}_S, \mathcal{C}_{TGG_S}) \subseteq \mathcal{L}(TG_S, \mathcal{C}_{TGG_S})$, and analogously $\mathcal{L}(\mathcal{R}_T, \mathcal{C}_{TGG_T}) \subseteq \mathcal{L}(TG_T, \mathcal{C}_{TGG_T})$. According to Def. 53, (TGG, \mathcal{C}_{TGG}) is, therefore, schema-compliant. \square

Example 40 (Schema-compliance and precedence-compatibility by construction). —

Figure 6.2 depicts the application of the construction technique to the TGG rule `FirstGotoRule` for the target negative constraint `NoMultiplePaint`. This process is repeated analogously for all rules and for all constraints. As only source and target negative constraints are supported, i.e., the conditions consist of a single graph P , the construction as presented in Fact 11 can be simplified by omitting the two bottom rectangles. For presentation purposes, some details such as names and types of correspondence links are omitted in Fig. 6.2.

The first step in the process is to determine all gluings of the right-hand side of the triple rule R and the negative constraint P . These gluings represent all possible ways in which the constraint can be violated by applying the rule. In the concrete example depicted in Fig. 6.2, there are only two possibilities: ① the trivial gluing, i.e., gluing no nodes of R and P , and ② gluing the Commands c_0 and c together.

The next step in the process is to determine the state of the graph *before* rule application by deleting all elements that are created by the rule. The resulting triple graph P' (the pushout complement of the direct derivation) represents a state of the graph that would lead to a constraint violation via rule application and must thus be forbidden as a NAC.

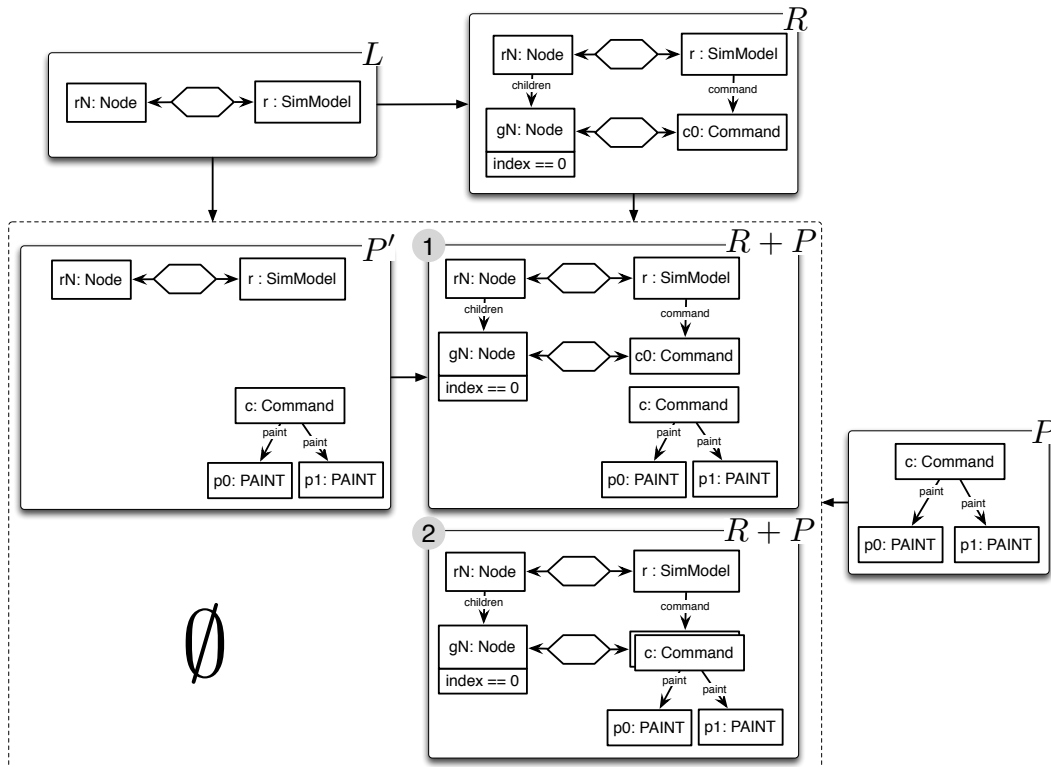


Figure 6.2: Constructing NACs for FirstGotoRule and NoMultiplePaint

In case ①, P' simply states an obvious fact: if the constraint was already violated *before* rule application then it is still violated after rule application (TGG rules are non-deleting and can never correct violations of negative constraints). As this is always the case, we shall demand in the rest of this chapter that the target graph be schema-compliant before rule application and, therefore, ignore such default gluings.

Case ② demonstrates that the pushout complement P' does not always exist! In this case, deleting the elements created by the rule would lead to dangling paint edges. It is, therefore, impossible to violate the constraint via this gluing.

If the input triple graph is required to be schema-compliant, the result of applying the construction technique shows that it is impossible to violate NoMultiplePaint via applications of FirstGotoRule. Intuitively, this result is not surprising as FirstGotoRule creates a new Command without any Paints.

Note also that the process is applied component-wise; violations of a *target* negative constraints can always be prevented with a *target* NAC.

To construct a NAC that is actually necessary for ensuring schema-compliance, the process is repeated again for FirstGotoRule and this time the source negative constraint NoMultipleFirstGoto. For presentation purposes, only the source components of the constructed triple graphs are depicted in Fig. 6.3.

The following gluings of R and P are possible here: the default gluing (not shown as we assume schema-compliance of the source graph), ① gluing n and rN as well as $n1$ and gN , ② only gluing n and rN , and ③ only gluing $n1$ and

gN. Note that in cases ① and ③, the results of gluing n_0 instead of n_1 produce isomorphic results and are not depicted explicitly.

In case ①, the constructed P' corresponds to the NAC in the corrected version of FirstGotoRule already discussed in Ex. 33 as FirstGotoRule::V2 (Fig. 5.23). In case ②, P' also exists but can be ignored as it is redundant in the following sense: P' constructed in ① can be embedded injectively in P' constructed in ②. This means that every violation of ② is also a violation of ①, i.e., ① is “stronger” than ②. In case ③, P' cannot be constructed as no pushout complement exists. Intuitively, the resulting NAC makes sense, FirstGotoRule can only violate NoMultipleFirstGoto, if the root node already has exactly such a goto node (or of course *multiple* such goto nodes).

An interesting point to note is that the constructed P' in case ② also violates the constraint! This means that even if there were no stronger NAC, we would still discard the constructed NAC as the required schema-compliance of the source graph already prevents this situation.

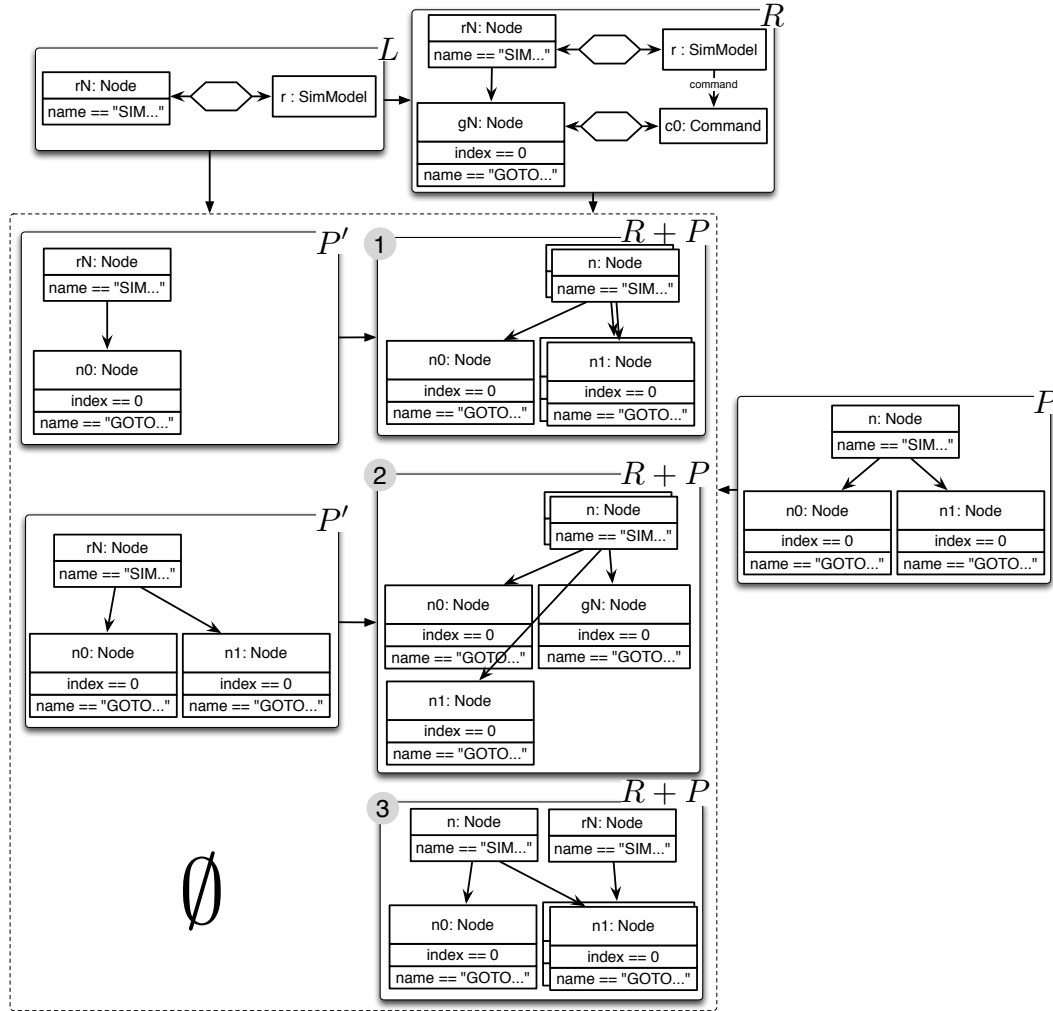


Figure 6.3: Constructing NACs for FirstGotoRule and NoMultipleFirstGoto

On a final note, the presented construction technique indeed guarantees operationalizability, schema-compliance, and precedence-compatibility, but poses a series of challenges on an implementation as demonstrated by this example: (i) symmetries of R and P must be exploited to avoid generating isomorphic NACs, (ii) a weaker/stronger relationship between NACs must be used to eliminate redundant NACs, and (iii) generated NACs that already violate *any* constraint must also be discarded as they cannot occur in schema-compliant input graphs.

6.2 STATIC ANALYSIS OF SOURCE LOCAL COMPLETENESS

Precedence matches are ideal for updating a precedence graph as they can be determined independently of each other and in any order. Precedence relations only have to be collected, updated from the match, and used to correct or extend the existing precedence graph.

For static analyses, however, it is easier to reason about source and forward rules that demand a certain *marking* of the triple graph and extend this *marking* via rule application. Context elements are demanded to be already marked, while “created” elements are marked when the rules are applied.

Definition 63 formalizes this idea for source/target marking rules. Note that a single marker node type $\overline{\text{tr}}$ is introduced for “markers”. This means that every marked element, node $v : \bar{v}$ or edge $e : \bar{e}$, gets its own marker node $v_{\text{tr}} : \overline{\text{tr}}$ or $e_{\text{tr}} : \overline{\text{tr}}$, respectively.

Using these marker nodes, a *translated* graph can be represented in a containing graph by connecting all its elements to marker nodes. Translated nodes are marked with v_{tr} marker nodes, while translated edges are marked with e_{tr} marker nodes.

Marker edge types $\bar{e}_{\bar{v}}$ are used to connect node markers v_{tr} to their marked nodes v of type \bar{v} , while edges e_s and e_t must be used to “connect” an edge marker e_{tr} to its edge e , as our graph model does not allow connecting nodes (markers) directly to edges. Although multiple edges of the same type between the same two nodes are theoretically possible and connect be uniquely marked in this manner, this is not of practical relevance (for this thesis) and is assumed to be forbidden with appropriate constraints.

Definition 63 (Source Marking Rules).

Given an operationalizable $\text{TGG} = (\text{TG}_S \xrightarrow{\sigma_{\text{TG}}} \text{TG}_C \xrightarrow{\tau_{\text{TG}}} \text{TG}_T, \mathcal{R})$, the *source type graph with translation markers* $\text{TG}_S^* = \text{tr}(\text{TG}_S)$ is defined as follows:

$$\begin{aligned} V_{\text{TG}_S^*} &:= V_{\text{TG}_S} \cup \{ \overline{\text{tr}} \} \\ E_{\text{TG}_S^*} &:= E_{\text{TG}_S} \cup \{ \bar{e}_{\bar{v}} : \overline{\text{tr}} \rightarrow \bar{v} \mid \text{for every } \bar{v} \in V_{\text{TG}_S} \} \end{aligned}$$

Given typed source graphs $G_S, H_S \in \mathcal{L}(\text{TG}_S)$, $H_S \subseteq G_S$,
the *translated graph* $H_S^* = \text{tr}_{G_S}(H_S)$ for H_S in G_S is defined as follows:

$$\begin{aligned} V_{H_S^*} &:= V_{G_S} \\ &\cup \{v_{\text{tr}} \mid \text{for every } v \in V_{H_S}, \text{type}(v_{\text{tr}}) = \overline{\text{tr}}\} \\ &\cup \{e_{\text{tr}} \mid \text{for every } e \in E_{H_S}, \text{type}(e_{\text{tr}}) = \overline{\text{tr}}\} \\ E_{H_S^*} &:= E_{G_S} \\ &\cup \{e_v : v_{\text{tr}} \rightarrow v \mid v \in V_{H_S}, \text{type}(e_v) = \overline{e}_{\text{type}(v)}\} \\ &\cup \{e_s : e_{\text{tr}} \rightarrow \text{src}(e) \mid e \in E_{H_S}, \text{type}(e_s) = \overline{e}_{\text{type}(\text{src}(e))}\} \\ &\cup \{e_t : e_{\text{tr}} \rightarrow \text{trg}(e) \mid e \in E_{H_S}, \text{type}(e_t) = \overline{e}_{\text{type}(\text{trg}(e))}\} \end{aligned}$$

Given a source rule $\text{sr} : [L_S \leftarrow \emptyset \rightarrow \emptyset = \text{SL}] \rightarrow [R_S \leftarrow \emptyset \rightarrow \emptyset = \text{SR}]$ with precedence conditions $\mathcal{PC}_S(\text{sr})$, the *source marking rule* $\text{sr}^* : \text{SL}^* \rightarrow \text{SR}^*$, with $\mathcal{PC}_S(\text{sr})$ and \mathcal{N}_{tr} as conditions over SL^* , is defined as follows:

$$\begin{aligned} \text{SL}^* &:= L_S^* \leftarrow \emptyset \rightarrow \emptyset, \text{ where } L_S^* = \text{tr}_{R_S}(L_S) \\ \text{SR}^* &:= R_S^* \leftarrow \emptyset \rightarrow \emptyset, \text{ where } R_S^* = \text{tr}_{R_S}(R_S) \\ \mathcal{N}_{\text{tr}} &:= \{n_v : L_S^* \rightarrow N_v \mid v \in V_{R_S} \setminus V_{L_S}\} \\ &\cup \{n_e : L_S^* \rightarrow N_e \mid e \in E_{R_S} \setminus E_{L_S}\}, \end{aligned}$$

where (with appropriately extended source and target functions):

$$\begin{aligned} V_{N_v} &:= V_{L_S} \cup \{v_{\text{tr}}, \text{type}(v_{\text{tr}}) = \overline{\text{tr}}\} \\ E_{N_v} &:= E_{L_S} \cup \{e_v : v_{\text{tr}} \rightarrow v, \text{type}(e_v) = \overline{e}_{\text{type}(v)}\} \\ V_{N_e} &:= V_{L_S} \cup \{e_{\text{tr}}, \text{type}(e_{\text{tr}}) = \overline{\text{tr}}\} \\ E_{N_e} &:= E_{L_S} \cup \{e_s : e_{\text{tr}} \rightarrow \text{src}(e), \text{type}(e_s) = \overline{e}_{\text{type}(\text{src}(e))}\} \\ &\cup \{e_t : e_{\text{tr}} \rightarrow \text{trg}(e), \text{type}(e_t) = \overline{e}_{\text{type}(\text{trg}(e))}\} \end{aligned}$$

Target marking rules are defined analogously.

A source marking rule is defined by taking the corresponding source rule, extending left and right-hand sides with translation markers, taking all precedence conditions, and adding a set of NACs that collectively demand the required translation state of the left-hand side. This is illustrated further with the following example. Note that, for presentation purposes, a *target* marking rule is discussed instead of a source marking rule as the target fragments of the rules of the running example are more compact.

Example 41 (Target marking rule for GotoPaintXYTargetRule). —————

In this and following examples, we shall investigate the violation of source local completeness discussed in Ex. 37. The problematic rules were a pair of paint rules `GotoPaintXYRule` and `GotoPaintXYZRule`.

Figure 6.4 depicts the translated graph $\text{tr}_{\text{TR}}(\text{TL})$ for the target rule $\text{GotoPaintXYTargetRule} : \text{TL} \rightarrow \text{TR}$, to the left in a detailed formal syntax according to Def. 63, and for presentation purposes, to the right in a compact syntax according to [67]. In the compact syntax, used in the rest of this example, the marker elements are visualized as filled “checkboxes” placed beside the marked node or edge, and trivial components are not shown.

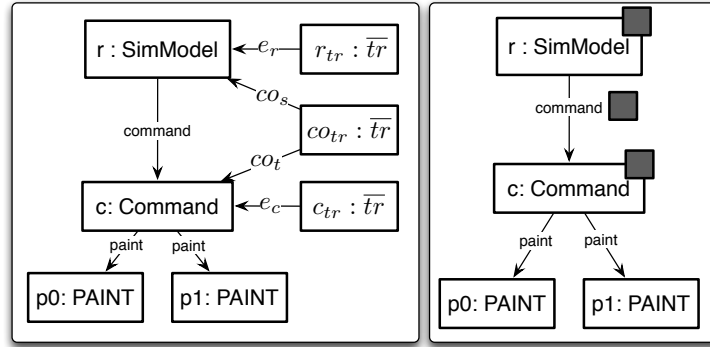


Figure 6.4: Translated graph $\text{TL}^* = \text{tr}_{\text{TR}}(\text{TL})$ in detailed and compact notation

This compact syntax is also used to represent marking rules such as the target marking rule for $\text{GotoPaintXYTargetRule}$ depicted in Fig. 6.5. In this figure, *empty* checkboxes are used to indicate the translation NACs that forbid the presence of marker elements for the respective elements. Elements (nodes and edges) with filled checkboxes must be marked, while elements with empty checkboxes must *not* be marked for the marking rule to be applicable. Note that only the non-trivial components of the marking rules (in this case the target components) are depicted.

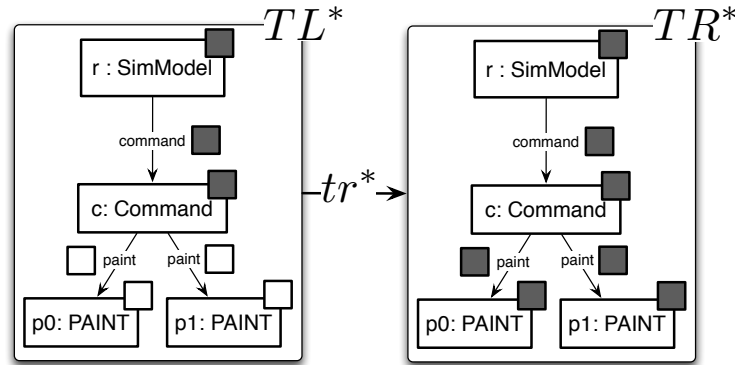


Figure 6.5: Target marking rule for GotoPaintXYRule in compact notation

To be able to apply further well-known static analysis techniques, source marking rules are extended to forward marking rules by replacing their trivial components with the components from the corresponding forward rule. A forward marking rule, therefore, not only marks the source graph but also creates all new elements in the correspondence and target domains. This is formalized with the following definition.

Definition 64 (*Forward Marking Rules*).

Let $r : (L_S \xleftarrow{\sigma_L} L_C \xrightarrow{\tau_L} L_T) \rightarrow (R_S \xleftarrow{\sigma_R} R_C \xrightarrow{\tau_R} R_T) \in \mathcal{R}$ be a TGG rule.

Its *forward marking rule* $fr^* : FL^* \rightarrow FR^*$ is defined as follows:

$$\begin{aligned} FL^* &:= tr_{R_S}(L_S) \leftarrow L_C \rightarrow L_T \\ FR^* &:= tr_{R_S}(R_S) \leftarrow R_C \rightarrow R_T \end{aligned}$$

with all conditions from the source marking rule sr^* of r .

Backward marking rules are defined analogously.

To be able to transfer the results of static analyses performed on source and forward marking rules to the actual precedence matches used for synchronization, the following lemma guarantees two things: (a) every precedence-induced source derivation corresponds to a derivation of source marking rules and vice-versa, and (b) every precedence-induced source derivation together with a match consistent derivation of forward rules corresponds to a derivation of forward marking rules and vice-versa.

Lemma 7 (*Equivalence of Precedence-Induced and Marking Derivations*).

Let $TGG = (TG, \mathcal{R})$ be an operationalizable, precedence-compatible triple graph grammar, $TG_S^* = tr(TG_S)$ the source type graph with translation markers for TG_S , \mathcal{R}_S the set of source rules, \mathcal{R}_S^* the set of source marking rules, \mathcal{R}_F the set of forward rules, \mathcal{R}_F^* the set of forward marking rules, and PG_S the source precedence graph for $G_{S_n} \in \mathcal{L}(TG_S)$.

With $G_i := G_{S_i} \leftarrow G_{C_i} \rightarrow G_{T_i}$, $G_i^* := tr_{G_{S_n}}(G_{S_i}) \leftarrow G_{C_i} \rightarrow G_{T_i}$,

$sr_i \in \mathcal{R}_S$, $sr_i^* \in \mathcal{R}_S^*$, $fr_i \in \mathcal{R}_F$, $fr_i^* \in \mathcal{R}_F^*$, the following equivalences hold:

$$\begin{aligned} \text{(a)} \quad & \exists \text{ precedence-induced } G_{00} \xrightarrow{sr_1 @ sm_1} G_{10} \xrightarrow{sr_2 @ sm_2} \dots \xrightarrow{sr_i @ sm_i} G_{i0} \Leftrightarrow \\ & \exists G_{00}^* \xrightarrow{sr_1^* @ sm_1^*} G_{10}^* \xrightarrow{sr_2^* @ sm_2^*} \dots \xrightarrow{sr_i^* @ sm_i^*} G_{i0}^* \\ \text{(b)} \quad & \exists \text{ precedence-induced } G_{00} \xrightarrow{sr_1 @ sm_1} G_{10} \xrightarrow{sr_2 @ sm_2} \dots \xrightarrow{sr_i @ sm_i} G_{i0} \\ & \text{and } \exists \text{ match consistent } G_{i0} \xrightarrow{fr_1 @ fm_1} G_{i1} \xrightarrow{fr_2 @ fm_2} \dots \xrightarrow{fr_i @ fm_i} G_{ii} \Leftrightarrow \\ & \exists G_{00}^* \xrightarrow{fr_1^* @ fm_1^*} G_{11}^* \xrightarrow{fr_2^* @ fm_2^*} \dots \xrightarrow{fr_i^* @ fm_i^*} G_{ii}^* \end{aligned}$$

Proof.

$$\begin{aligned} \text{(a)} \quad & \exists \text{ precedence-induced } G_{00} \xrightarrow{sr_1 @ sm_1} G_{10} \xrightarrow{sr_2 @ sm_2} \dots \xrightarrow{sr_i @ sm_i} G_{i0} \xrightarrow{\text{Def. 63}} \\ & G_{i0} \xrightarrow{sr_i @ sm_i} G_{i+1,0} \text{ corresponds to } G_{i0}^* \xrightarrow{sr_i^* @ sm_i^*} G_{i+1,0}^* \\ & \text{where } sr_i^* : SL^* \rightarrow SR^* \text{ is the source marking rule of } sr : SL \rightarrow SR, \text{ and} \end{aligned}$$

$$\begin{aligned}
& sm_i^* : SL^* \rightarrow G_{i0}^* \text{ is constructed analogously from } sm : SL \rightarrow G_{i0} \xLeftrightarrow{\text{Def. 59}} \\
& sm_i \models \mathcal{PC}(sr) \Leftrightarrow sm_i^* \models \mathcal{PC}(sr) \xLeftrightarrow{\text{Def. 59}} \\
& \forall i \neq j, \text{created}(v_i) \cap \text{created}(v_j) = \emptyset \Leftrightarrow sm_i^* \models \mathcal{N}_{tr} \Leftrightarrow \\
& \exists G_{00}^* \xRightarrow{sr_1^* @ sm_1^*} G_{10}^* \xRightarrow{sr_2^* @ sm_2^*} \dots \xRightarrow{sr_i^* @ sm_i^*} G_{i0}^* \\
& \text{(b) } \exists \text{ precedence-induced } G_{00} \xRightarrow{sr_1 @ sm_1} G_{10} \xRightarrow{sr_2 @ sm_2} \dots \xRightarrow{sr_i @ sm_i} G_{i0} \text{ and} \\
& \exists \text{ match consistent } G_{i0} \xRightarrow{fr_1 @ fm_1} G_{i1} \xRightarrow{fr_2 @ fm_2} \dots \xRightarrow{fr_i @ fm_i} G_{ii} \xLeftrightarrow{\text{Thm. 6}} \\
& \exists G_{00} \xRightarrow{r_1 @ m_1} G_{11} \xRightarrow{r_2 @ m_2} \dots \xRightarrow{r_i @ m_i} G_{ii} \xLeftrightarrow{\text{Lemma 7(a), Def. 64}} \\
& \exists G_{00}^* \xRightarrow{fr_1^* @ fm_1^*} G_{11}^* \xRightarrow{fr_2^* @ fm_2^*} \dots \xRightarrow{fr_i^* @ fm_i^*} G_{ii}^*
\end{aligned}$$

□

Having established the concept of marking rules, we can now introduce the notion of *confluence*,¹ a well-known property of graph grammars that can be checked statically. Every partial derivation of a confluent graph grammar can be completed to a derivation that produces the same result, and this is exactly what is to be guaranteed by source local completeness, i.e., that every precedence-induced source derivation (corresponding to a partial marking of the source graph) constructed by trans can be completed to a complete precedence-induced source derivation (corresponding to a complete marking) of the input source graph. The following definition, taken from [28], formalizes this notion of confluence for graph grammars.

Definition 65 (Confluence).

A pair $P_1 \xLeftarrow{*} K \xRightarrow{*} P_2$ of derivations in a graph grammar is *confluent* if there exists an X together with derivations $P_1 \xRightarrow{*} X$ and $P_2 \xRightarrow{*} X$.

A graph grammar is *confluent* if all pairs of its derivations are confluent.

To ensure confluence statically, it is sufficient (but not necessary!) to demand that there exist no pairwise “conflicts” between possible direct derivations of the graph grammar. Although there are infinitely many direct derivations, it suffices to check all *critical pairs*, which are in a sense minimal. The following definition, taken from [69] and simplified for TGGs, formalizes the two relevant types of conflicts for source marking rules: the first direct derivation forbids an element that the second creates (*forbid-produce*), and vice-versa (*produce-forbid*).

Definition 66 (Critical Pair).

A critical pair $H_1 \xLeftarrow{r_1 @ m_1} G \xRightarrow{r_2 @ m_2} H_2$ of a graph grammar $GG = (TG, \mathcal{R})$ is a pair of direct derivations via rules $r_1 : L_1 \rightarrow R_1, r_2 : L_2 \rightarrow R_2 \in \mathcal{R}$ with sets of NACs $\mathcal{N}_1, \mathcal{N}_2$, respectively, such that:

1. $\exists n_1 : L_1 \rightarrow N_1 \in \mathcal{N}_1, \exists q_1 : N_1 \rightarrow H_2 \in \mathcal{M}, r_2' \circ m_1 = q_1 \circ n_1$ **or**

¹ From Latin: *confluere* “flow together” (Oxford English Dictionary).

$$2. \exists n_2 : L_2 \rightarrow N_2 \in \mathcal{N}_2, \exists q_2 : N_2 \rightarrow H_1 \in \mathcal{M}, r'_1 \circ m_2 = q_2 \circ n_2$$

(1) and (2) are referred to as *forbid-produce*, and *produce-forbid* conflicts, respectively, and are depicted visually in Fig. 6.6

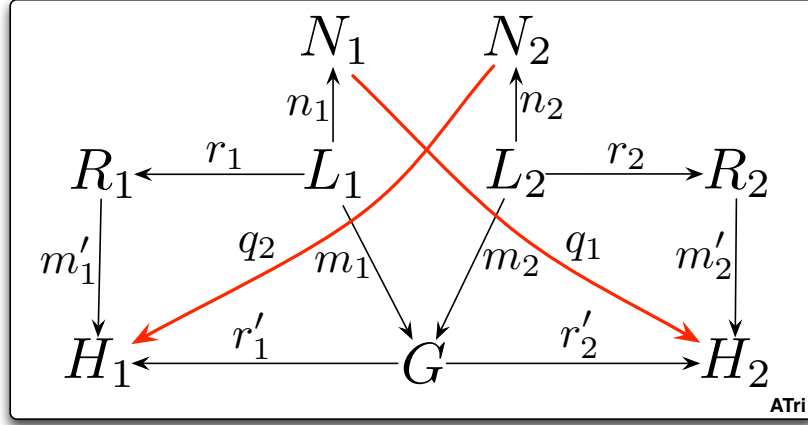


Figure 6.6: Forbid-produce and produce-forbid conflicts

Existing theory, stated in the following fact, provides a sufficient condition for confluence, namely that all critical pairs (potential conflicts) are confluent for a graph grammar.

Fact 12 (*Sufficient Condition for Confluence*).

A graph grammar GG with rules \mathcal{R} with NACs is confluent if all its critical pairs are confluent.

Proof. The interested reader is referred to [69] for a detailed proof in a more general setting. \square

The following corollary transfers these well-known results to source local completeness, stating that source local completeness of a TGG corresponds to confluence of the set of source marking rules of the TGG.

The main idea is to demand confluence of the *source marking rules* of a TGG. This guarantees that a given source graph can be marked without running into dead-ends. This is transferred to precedence induced derivations of the source graph using Lemma 7, implying source local completeness.

Note that source marking rules are equipped with all precedence conditions, i.e., DEC NACs and attribute conditions are all used to reduce conflicts in the sense of Def. 66. In general, however, the end user still has to inspect critical pairs and decide if the depicted situations are really potential violations of source local completeness as the condition is sufficient but not necessary.

Corollary 2 (*Sufficient Condition for Source Local Completeness*).

A triple graph grammar TGG is source locally complete if all critical pairs of its set \mathcal{R}_S^* of source marking rules are confluent.

Proof. All critical pairs of \mathcal{R}_S^* are confluent $\xRightarrow{\text{Fact. 12}} \mathcal{R}_S^*$ is confluent $\xRightarrow{\text{Def. 65}} \forall G_{S_n}^* \in \mathcal{L}(\mathcal{R}_S^*)$ every derivation $G_{S_0}^* \xRightarrow{*} G_{S_i}^*$ in \mathcal{R}_S^* can be completed to a derivation $G_{S_0}^* \xRightarrow{*} G_{S_n}^*$ in \mathcal{R}_S^* , where $G_{S_i}^* = \text{tr}_{G_{S_n}}(G_{S_i}) \leftarrow G_{C_i} \rightarrow G_{T_i}$.
 $\xRightarrow{\text{Lemma 7(a)}} \text{every precedence-induced derivation } G_{S_0} \xRightarrow{*} G_{S_i} \text{ in } \mathcal{R}_S \text{ can be completed to a precedence-induced derivation } G_{S_0} \xRightarrow{*} G_{S_n} \xRightarrow{\text{Def. 60}} \mathcal{R}_S \text{ is locally complete} \xRightarrow{\text{Def. 60}} \text{TGG is source locally complete.} \quad \square$

Example 42 (*A critical pair constructed from the problematic paint rules*). —————

We are now able to statically analyse the version of our running example with the pair of problematic paint rules introduced in Ex. 37.

According to Cor. 2, the TGG with `GotoPaintXYRule` and `GotoPaintXYZRule` is *not* source locally complete as the critical pair depicted in Fig. 6.7 (using the same labels as in Fig. 6.6 for readability) can be constructed. Note that r_1, r_2 correspond to `GotoPaintXYZRule` and `GotoPaintXYZRule`, respectively, and that N_1 and N_2 are only depicted explicitly for presentation purposes, i.e., there are NACs enforcing all other empty checkboxes for each rule application.

Intuitively, applications of `GotoPaintXYRule` and `GotoPaintXYZRule` can be in conflict as they can potentially mark the same elements and thus “compete” for these elements. This means that, depending on which application “wins”, a different result might be produced and, in the worst case, a wrong local choice might be made leading a dead-end, i.e., a partial marking that cannot be extended to a complete marking of the source graph. In such cases, trans can make a wrong decision.

Note that DEC NACs do not prevent the conflict in this case, as only context and creation patterns of *single links* are analysed for creating DEC NACs and not complete rule patterns. In this example, Alg. 5 will not identify applications of `GotoPaintXYRule` as being potentially dangerous, as the resulting state for the third paint link (translated Command, paint link and PAINT to be translated) is handled by both `GotoPaintXYRule` and `GotoPaintXYZRule`. This demonstrates that the check in Alg. 5 is only sufficient but not necessary and certainly reduces but does not prevent all possible conflicts. Extending Alg. 5 to take, for example, larger patterns into account is possible but will certainly have an adverse effect on efficiency as substantially larger NAC patterns must be checked for rule applicability.

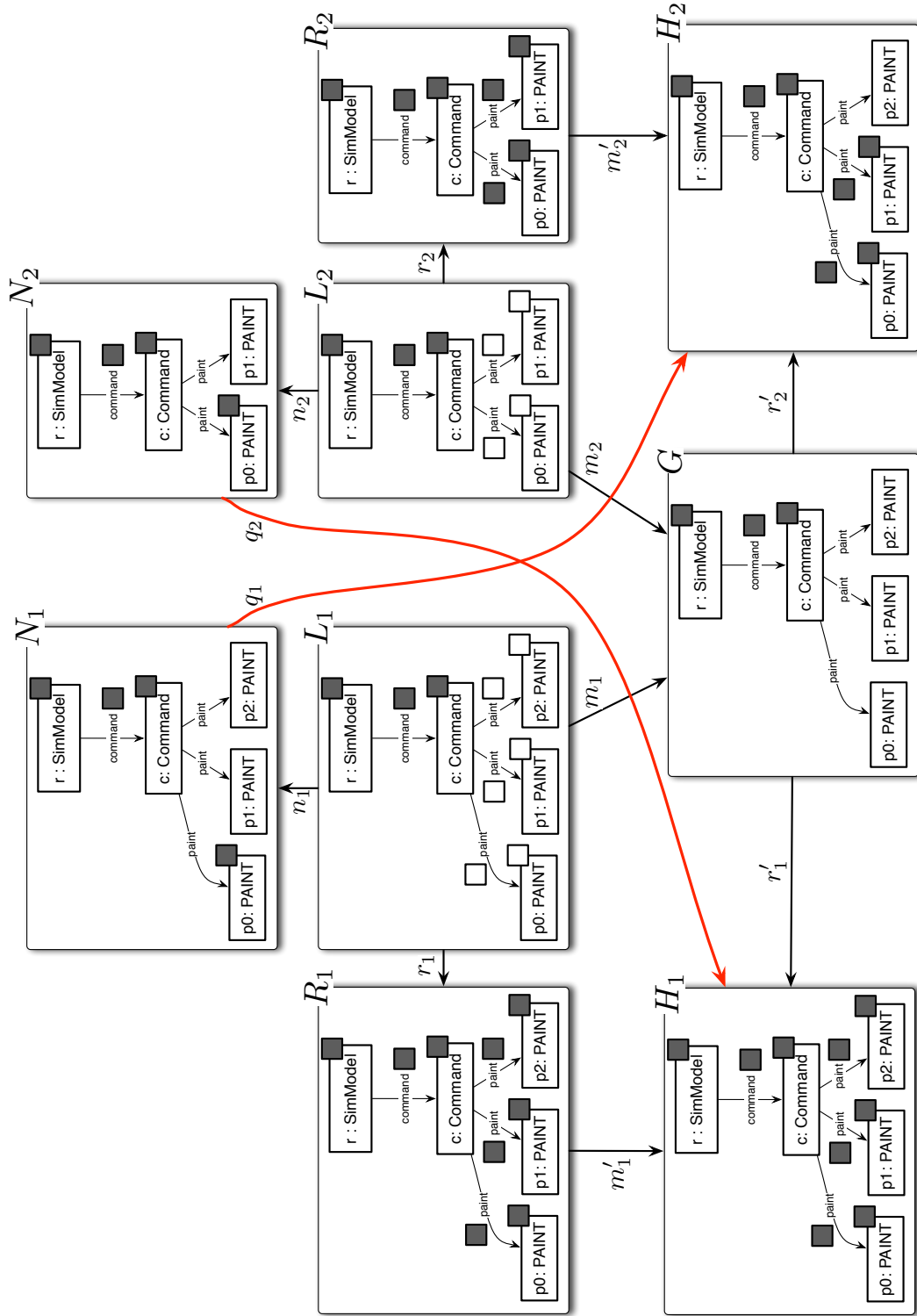


Figure 6.7: Critical pair for GotoPaintXYRule and GotoPaintXYZRule target marking rules

6.3 STATIC ANALYSIS OF FORWARD LOCAL COMPLETENESS

Intuitively, forward local completeness guarantees that the source precedence graph has enough information to control the forward synchronization process without making wrong choices between applicable rules. The main reason for demanding forward local completeness is to avoid backtracking in sync (Alg. 9).

The core idea that leads to a sufficient condition for forward local completeness, is to formulate the required property as a set of *forward local completeness constraints* according to Def. 9. If we succeed in doing this, then we can use the construction of Fact 11 to transform these constraints to application conditions for a given TGG. If Fact 11 generates trivial application conditions only, then the TGG is forward locally complete as required. This can be statically checked at rule specification time!

This approach has been formalized and presented in [8] for TGGs *without* NACs, attribute conditions or dynamic conditions. The rest of this chapter presents these results, generalized appropriately to cover NACs, attribute conditions, and dynamic conditions, but with certain limitations that require an extension of Fact 11 and Def. 9 to graphs with attribute constraints, i.e., using attribute conditions as constraints in graphs and not only as application conditions. This is, however, currently work in progress [24] and is out-of-scope for this thesis.

The main idea is to construct these constraints by taking the context of source marking rules as premise, and demanding the context of the corresponding forward marking rule as conclusion. If this constraint holds for the TGG, then all source derivations can be extended to complete derivations, which is what is required for forward local completeness.

The following definition formalizes this notion of a *forward local completeness constraint*, which captures the requirement of forward local completeness in a manner that is amenable to static analyses (e.g., with Fact 11).

Definition 67 (*Forward Local Completeness Constraint*).

Let $TGG = (TG, \mathcal{R})$ be an operationalizable triple graph grammar, and $\mathcal{R}_S^*, \mathcal{R}_F^*$ its sets of source marking and forward marking rules, respectively.

For all source marking rules $sr_1^* : SL^* \rightarrow SR_1^*, \dots, sr_n^* : SL^* \rightarrow SR_n^* \subseteq \mathcal{R}_S^*$, with corresponding forward marking rules $fr_1^* : FL_1^* \rightarrow FR_1^*, \dots, fr_n^* : FL_n^* \rightarrow FR_n^* \subseteq \mathcal{R}_F^*$ the *forward local completeness constraint* $flcc(SL^*)$ is defined as:

$$flcc(SL^*) := \forall c_i : SL^* \rightarrow FL_i^*$$

The set $flcc(SL^*)$ of forward local completeness constraints for TGG is defined as:

$$flcc(TGG) := \{flcc(SL^*) \mid sr^* : SL^* \rightarrow SR^* \in \mathcal{R}_S^*\}$$

The set of *backward local completeness constraints* is defined analogously.

Each forward local completeness constraint states that the context of a forward marking rule must follow from the context of the corresponding source marking rule. This is a positive constraint, i.e., an *if-then* constraint. As forward rules can be identical with respect to the source component, these rules must be merged via a disjunction to form a single forward local completeness constraint; given the context of the source marking rule, *one of the alternatives* must follow.

Note that it is crucial to formulate forward local completeness constraints over *marking* rules. Indeed, the required context of the forward rule can only be implied from the marking (translation state) of the source context. This is best illustrated with a concrete example.

Example 43 (*Forward local completeness constraints*). —————

Figure 6.8 depicts the *backward* local completeness constraint derived from `GotoPaintRule::V0`. As `GotoModalPaintRule::V0` does not differ from `GotoPaintRule::V0` in the target component (TL^*), the constraint is a disjunction, demanding the context of the backward rule of `GotoPaintRule::V0` ①, or the context of the backward rule of `GotoModalPaintRule::V0` ②. Intuitively, the constraint expresses the following:

During the translation process, if a simulator model and a connected command are translated, then the corresponding root node and goto node are present in the tree, or there might even additionally be a last non-trivial colour node (lntcolN).

In this case the constraint c_1 is always fulfilled if c_2 is fulfilled, expressing the fact that it should *always* be possible to translate a PAINT command normally, and in some cases, there might be a freedom of choice to translate it modally.

Figure 6.9 depicts the forward local completeness constraint derived from `PaintGotoRule::V0`. In this case, the constraint expresses the following:

During the translation process, if a simulator model and a connected command are translated, then the corresponding root node and goto node, as well as a sibling paint node, are present in the tree.

This example is interesting as it reveals a problem with `GotoPaintRule::V0` but also a severe (current) limitation of the approach. The problem is that the constraint is obviously unreasonable: translating a command does not in any way guarantee that there is a single paint node in the tree. This indicates that target context information is missing in `GotoPaintRule::V0` to imply the existence of the required paint node.

The limitation of the approach is that the constraint only demands the existence of *a* sibling paint node in the tree. This is not precise; what is actually required by `GotoPaintRule::V0` is a paint node that is a *direct sibling* of the goto node (gN) in the tree. Analogously to the attribute application condition in the rule, this must be expressed with an attribute constraint `add(gN.index, 1, pN.index)`.

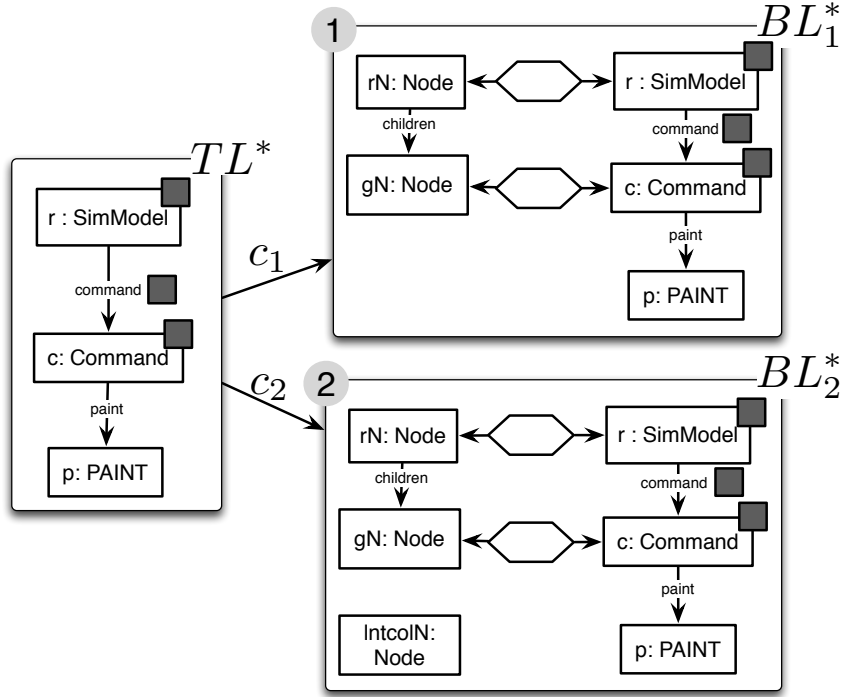


Figure 6.8: Forward local completeness constraint for $\text{GotoPaintRule}::V0$

In this thesis, however, attribute conditions have only been introduced as application conditions for triple rules. Attribute *constraints* as required here are out-of-scope (cf. [24] for work in this direction) and Fact 11 cannot be applied. Even with this limitation, the constraint still shows that the TGG with $\text{GotoPaintRule}::V0$ is not forward locally complete.

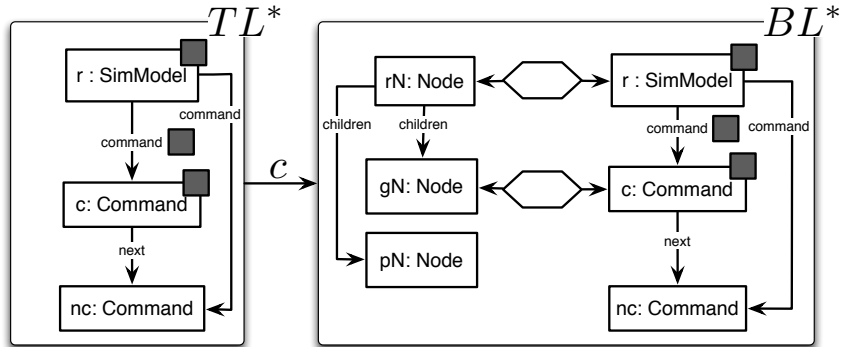


Figure 6.9: Forward local completeness constraint for $\text{PaintGotoRule}::V0$

Can we express precisely when Fact 11 is sufficient and when not? We can do this by demanding that the guarantees provided by enforcing forward local completeness constraints are not contradicted by attribute conditions, dynamic conditions, or NACs. Handling NACs first, the required condition is that target NACs must not contradict forward local completeness constraints. This gives rise to the following notion of *forward redundancy* of target NACs. As NACs are *only* used to

ensure schema-compliance (this is demanded by precedence-compatibility), a target NAC is *forward redundant* if it blocks violations of a target constraint, which is already ensured by fulfilling a corresponding source constraint. This means that if the source graph is schema-compliant, then the target constraint cannot be violated and the target NAC can be deleted in the forward rule. This condition can be checked by demanding that the TGG does not produce triples, which violate the target constraint without violating the source constraint. The following definition formalizes these unproblematic (forward redundant) target NACs.

Definition 68 (*Forward Redundant Negative Target Constraints and Target NACs*).

Let $TGG = (TG, \mathcal{R})$ be a triple graph grammar without application conditions, and $\mathcal{C}_{TG_S}, \mathcal{C}_{TG_T}$ be sets of negative constraints over TG_S and TG_T , respectively.

A negative target constraint $NC_T \in \mathcal{C}_{TG_T}$ is *forward redundant* if every violation of NC_T by the target component of a triple implies a violation of a corresponding negative source constraint $NC_S \in \mathcal{C}_{TG_S}$ by the source component of the triple:

$$\begin{aligned} & \exists (NC_S \leftarrow NC_C \rightarrow NC_T) \in \mathcal{L}(TGG), NC_S \in \mathcal{C}_{TG_S}, \mathcal{R} \models c_{NC_T}, \\ & \text{where } c_{NC_T} : (\emptyset \leftarrow \emptyset \rightarrow NC_T) \rightarrow (NC_S \leftarrow NC_C \rightarrow NC_T). \end{aligned}$$

A target NAC N_T is *forward redundant* if it is only required to prevent a violation of a forward redundant constraint NC_T .

Backward redundant negative source constraints and *backward redundant* source NACs are defined analogously.

The following lemma states that source schema-compliance is sufficient to ensure that forward redundant target constraint are not violated, implying that the corresponding target NACs are not required in forward rules.

Lemma 8 (*Forward Redundancy*).

Let $TGG = (TG, \mathcal{R})$, and $\mathcal{C}_{TG_S}, \mathcal{C}_{TG_T}$ be sets of negative constraints over TG_S and TG_T , respectively.

$$\forall G \in \mathcal{L}(TGG), \quad NC_T \text{ forward redundant} \Rightarrow [G_S \models \mathcal{C}_{TG_S} \Rightarrow G_T \models NC_T]$$

Proof. NC_T forward redundant $\stackrel{\text{Def. 68}}{\Rightarrow} [\exists (NC_S \leftarrow NC_C \rightarrow NC_T) \in \mathcal{L}(TGG), NC_S \in \mathcal{C}_{TG_S}, \mathcal{R} \models (\emptyset \leftarrow \emptyset \rightarrow NC_T) \rightarrow (NC_S \leftarrow NC_C \rightarrow NC_T)] \Rightarrow$
 $[\exists nc_T : NC_T \rightarrow G_T \Rightarrow \exists nc_S : NC_S \rightarrow G_S] \Leftrightarrow$
 $[\nexists nc_S : NC_S \rightarrow G_S \Rightarrow \nexists nc_T : NC_T \rightarrow G_T] \stackrel{\text{Def. 9}}{\Rightarrow}$
 $[G_S \models \mathcal{C}_{TG_S} \Rightarrow G_T \models NC_T]. \quad \square$

The argument is that for a forward redundant target constraint NC_T , violations of NC_T always imply a violation of some source constraint NC_S . Conversely, if no source constraints are violated, i.e., $G_S \models \mathcal{C}_{TG_S}$, then the target constraint can not be violated, i.e., $G_T \models NC_T$.

The attentive reader should now be wondering why target NACs are useful if we only allow forward redundant NACs that can be deleted anyway. There are three reasons why redundant NACs are nonetheless useful.

The first reason is that TGG specifications are not only used for synchronization but also in other operational scenarios such as for test case generation [56, 99] where the rules of the TGG are applied directly; both source and target NACs are required to ensure that only schema-compliant triples are generated.

The second reason is that it is useful in practice to reject non schema-compliant source graphs. If a synchronizer derived from a locally complete TGG aborts, it is often because the source graph was not schema-compliant, and a target NAC has blocked a violation of a corresponding target constraint. This is used to provide useful feedback for the end-user.

The third and final reason is that another class of target NACs are used in practice to express “alternative rules”. Such NACs do not have to be redundant as there always exists a *complementary* rule that is applicable exactly when the NAC is violated (i.e., the forbidden elements are present in the target graph). Such NACs are often generated to handle, e.g., cases where a certain rule is to be used only for the first element in a chain, and another rule for all other elements. The former rule would have a NAC forbidding a previous element and thus blocking the application of the rule for all elements apart from the first element in the chain, while the latter rule demands exactly the forbidden previous element as context. This is formalized with the following definition.

Definition 69 (*Complementary Rules*).

Given a TGG rule $r : L \rightarrow R$ and sets N_S, N_T of source and target NACs.

A rule $\bar{r}_{N_T} : \bar{L} \rightarrow \bar{R}$ together with sets $\bar{N}_S = N_S, \bar{N}_T = \emptyset$ of source and target NACs is a *forward complementary* rule for $N_T \in N_T$, if (\bar{R}_C, \bar{R}_T) can be chosen freely):

$$\bar{L} = L_S \leftarrow L_C \rightarrow N_T, \bar{R} = R_S \leftarrow \bar{R}_C \rightarrow \bar{R}_T$$

Backward complementary rules are defined analogously.

Attribute and dynamic conditions can be handled analogously by extending the notion of forward redundancy appropriately. Attribute conditions on the source side (target side for backward local completeness constraints) are obviously unproblematic. Similarly, attribute conditions that do not range over *target context elements* are also unproblematic as they are solved in the source match, and the values are simply used to apply the rule. Consider, e.g., `stringToNumber(colour-Node.name, paint.colour)` in `GotoPaintRule::V1` (Fig. 5.22). Such constraints cannot contradict forward local completeness constraints.

Problematic attribute conditions, therefore, range over target context elements such as `add(prevGotoNode.index, 1, paintNode.index)` (Fig. 5.21). This is the reason why the derived backward local completeness constraint in the previous example can be contradicted as soon as there is *any* paint node in the tree, but not the *right* one as demanded by the attribute condition!

Forward redundancy for these problematic attribute conditions means that they can be deleted in the forward rule without changing semantics (the language generated by the forward rules). Surprisingly, this is often the case as the correspondence model can be used to restrict the match morphism to the “right” target elements. In this case, such attribute conditions are only necessary when coming from the source side (the tree), but not when coming from the target side (the model). Consider for example `PaintGotoRule::V2`, the corrected version of the rule (Fig. 5.37). The target component of the rule is extended by an extra constraint element `paint`. This makes the problematic attribute condition “backward redundant” as the “right” `paint` node in the tree is chosen with the unique correspondence element `paintNodeToPaint`; the attribute condition can thus be deleted in the backward rule. The following definition formalizes the notion of *forward redundancy* for attribute and dynamic conditions.

Definition 70 (*Forward Redundant Attribute and Dynamic Conditions*).

Let $TGG = (TG, \mathcal{R})$ be a triple graph grammar and $r : L \rightarrow R \in \mathcal{R}$ a TGG rule with target dynamic conditions and attribute conditions.

A target dynamic condition (i, β) of r with $i : I \rightarrow L$ is *forward redundant* if it can be eliminated from the forward rule fr of r without changing $\mathcal{L}(TGG_F)$, the language generated by forward rules.

Backward redundant source dynamic conditions are defined analogously.

An attribute condition $c(t_1, \dots, t_n)$ is *forward redundant* if it can be eliminated from the forward rule fr of r without changing $\mathcal{L}(TGG_F)$, the language generated by forward rules.

Backward redundant attribute conditions are defined analogously.

We are now ready to state a sufficient condition for forward local completeness. Basically, all forward local completeness constraints must be fulfilled and the results of applying Fact 11 must not be contradicted by attribute conditions, dynamic conditions, or NACs.

Theorem 9 (*Sufficient Condition for Forward Local Completeness*).

Given an operationalizable, schema-compliant and precedence-compatible triple graph grammar $TGG = (TG, \mathcal{R})$ with forward marking rules \mathcal{R}_F^* .

TGG is *forward locally complete* according to Def. 61 if the following holds:

1. $\mathcal{R}_F^* \models flcc(TGG)$ (cf. Def. 67).
2. For all $r \in \mathcal{R}$, all target dynamic conditions of r are forward redundant.
For all $r \in \mathcal{R}$ and every attribute condition $c(t_1, \dots, t_n)$ of r :
 $\exists t_i \in \{t_1, \dots, t_n\}, t_i$ is an attribute value in $L_T \Rightarrow c$ is forward redundant.
3. For every rule $r \in \mathcal{R}$ and target NAC $N_T \in \mathcal{N}_T$ of r , N_T is either forward redundant, or there exists a complementary rule \bar{r}_{N_T} for N_T .

Proof.

To prove Thm. 9, we have to show forward local completeness according to Def. 61 for an operationalizable, schema-compliant, and precedence-compatible TGG, for which all three conditions stated in Thm. 9 hold. These conditions are referred to as Assumption(1), Assumption(2), and Assumption(3) in the following, and will be used to infer forward local completeness starting from Def. 61:

Given maximal $V^* = \{G_i \xrightarrow{sr@sm} SG_{i+1} \text{ precedence-induced} \mid \bigcap_{v \in V^*} \text{created}(v) \neq \emptyset\}$.

Let $v = sr@sm \in V^* \xrightarrow{V^* \text{ maximal}} \text{Def. 61} [\bar{v} \text{ complementary rule of } v \Rightarrow \bar{v} \in V^*]$.

Let $sr^* : SL^* \rightarrow SR^*$ be the source marking rule of $sr : SL \rightarrow SR$.

$G_i \xrightarrow{sr@sm} SG_{i+1} \text{ precedence-induced} \xrightarrow{\text{Lemma 7(a)}} G_i^* \xrightarrow{sr^*@sm^*} SG_{i+1}^* \xrightarrow{\text{Assumption(1)}}$

$\exists fr^* : FL^* \rightarrow FR^* \in \mathcal{R}_F^*, \exists c : SL^* \rightarrow FL^*, \exists q^* : FL^* \rightarrow G_i^*, sm^* = q^* \circ c$ (Fig. 6.10).

$sm \models \mathcal{PC}(sr) \xrightarrow{\text{Assumption(2), Def. 64,70}}$

$G_i^* \xrightarrow{fr^*@q^*} G_{i+1}^* (q^* \text{ fulfils all attribute and dynamic conditions}).$

CASE 1: (No target NAC is violated)

$\xrightarrow{\text{Lemma 7(b)}} \exists \text{ match consistent } G_i \xrightarrow{sr@sm} SG_{i+1} \xrightarrow{fr@q} G_{i+1}.$

CASE 2: (A target NAC is violated)

$\exists n_T^* : FL^* \rightarrow N_T^* \in \mathcal{N}_T(fr^*), \exists q_{n_T}^* : N_T^* \rightarrow G_i^*, q_{n_T}^* \circ n_T^* = q^* \xrightarrow{\text{Lemma 8, } G_i \subseteq G_n \models \mathcal{C}_{TG_S}}$

n_T is not redundant $\xrightarrow{\text{Assumption(3)}} \exists r_{n_T} : L \rightarrow N_T$ complementary rule for n_T

$\xrightarrow{\text{Def. 69}} \exists G_i^* \xrightarrow{fr_{n_T}^*@q_{n_T}^*} G_{i+1}^*, fr_{n_T}^* : N_T^* \rightarrow \overline{FR}^* \text{ (cf. Fig. 6.10)} \xrightarrow{\text{Lemma 7(b)}}$

$\exists \text{ match consistent } G_i \xrightarrow{sr@sm} SG_{i+1} \xrightarrow{fr_{n_T}^*@q_{n_T}^*} G_{i+1}.$ □

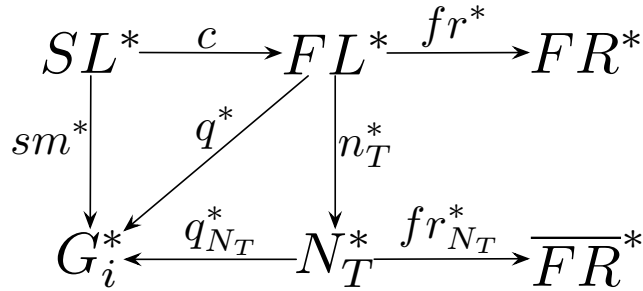


Figure 6.10: Situation used in proof of Thm. 9

Remark 4 (*Towards a Static Analysis of Local Completeness*).

A fully automated static analysis of the sufficient conditions for source local completeness (Cor. 2) and for forward local completeness (Thm. 9) is currently work in progress [24] and is out-of-scope for this thesis. The following presents the basic ideas towards a completely automated static analysis with the corresponding current limitations:

- The condition stated in Cor. 2 can be checked statically via the Critical Pair Analysis (CPA) of [69]. Note that conflicts between identical source rules must be ignored. In practice, the set of generated critical pairs must be further filtered to take constraints, attribute conditions, and dynamic conditions into account, as well as cases where the grammar is confluent regardless of the identified critical pair(s). This is a well-known limitation of CPA and is currently a manual process. Future work includes extending CPA to cover attribute conditions as well as implementing default filters, e.g., for critical pairs that violate constraints and are thus irrelevant if schema-compliance of the input source graph is assumed.
- Condition (1) of Thm. 9 can be checked statically by constructing the set $\mathcal{A}(\mathcal{R}^*, \text{flcc}(\text{TGG}))$ of application conditions according to Fact 11, required to guarantee $\mathcal{R}_F^* \models \text{flcc}(\text{TGG})$, and by demanding that this set consist of trivial application conditions only.

There are, however, two current limitations to this idea:

1. The constraint language used in this thesis is severely limited and must be extended to be able to formulate, e.g., nested constraints [51] to handle NACs in forward local completeness constraints.
2. Attribute conditions have only been used as application conditions in this thesis and not as part of the actual graphs themselves. For such transformations as in Fact 11, graphs with attribute constraints are required to be able to take all attribute conditions in the rules into account.

Due to these current limitations, Fact 11 only guarantees the existence of context elements, meaning that forward redundancy for all other conditions must be additionally demanded.

- Condition (2) of Thm. 9 can be checked statically by inspecting all attribute and dynamic conditions of all rules and checking for forward redundancy (Def. 70). This is currently done manually (typically by exploiting the correspondence model to avoid invalid matches) but could be automated via an appropriate extension of Fact 11. This is, however, out-of-scope for this thesis and is left to future work.

- Condition (3) of Thm. 9 can be checked statically by constructing the set $\mathcal{A}(\mathcal{R}, \{c_{NC_T}\})$ (cf. Fact 11) required to guarantee forward redundancy of NC_T and the corresponding target NACs (Def. 68), and by demanding that this set consist of trivial application conditions only. If this is not the case for some NAC N_T , then the existence of a complementary rule according to Def. 68 can be checked statically as a structural property of the TGG.

Similar limitations as for condition (1) hold here as well; the constraint language is often not expressive enough to formulate the source constraint required to show forward redundancy of a target NAC N_T .

Example 44 (Checking condition (1) for FirstGotoRule and the forward local completeness constraint of PaintGotoRule::V0).

To demonstrate the substantial potential of Fact 11 for an automated static analysis of forward local completeness, Fig. 6.11 depicts the process for the forward local completeness constraint derived from `PaintGotoRule::V0` applied to the rule `FirstGotoRule`.

The first step in the process is to construct all gluings $R + P$ of the right-hand side of the rule R , and the premise P of the forward local completeness constraint. The next step is to construct the unique pushout D . Now all possible gluings of *elements in* D are constructed by choosing non-injective morphisms $D \rightarrow D_i$. One of these gluings D_1 is depicted in Fig. 6.11.

In a final step, the premise P' and conclusion C_i of the application condition are constructed as pushout complements. These might not exist for some gluings or $P' = C_i$ might hold, meaning that the application condition is trivially satisfied. In Fig. 6.11, however, the gluings have been chosen so that a non-trivial application condition C_1 is indeed generated, showing that the rule `FirstGotoRule` can violate the forward local completeness constraint and that the TGG is not forward locally complete. The generated application condition states the following:

If a translated simulator model has a command with a next command, then the corresponding root node must have a paint node.

Note the subtlety of this positive application condition: `FirstGotoRule` can only violate the constraint if the matched command has a next command. If it does not then the premise of the constraint P cannot hold. This example demonstrates the potential for automating this check as a static analysis, but also indicates the numerous challenges that an implementation faces including: (i) dealing efficiently with the large number of possible gluings, (ii) presenting results in a helpful manner for the end-user, and (iii) filtering out generated but trivial or redundant application conditions.

The TGG with the corrected versions of the rules `PaintGotoRule::V2` and `GotoPaintModalRule::V2` (Fig. 5.37) is backward locally complete as all generated application conditions are trivial, and all problematic attribute conditions and dynamic conditions are backward redundant. Considering the source NAC

in FirstGotoRule (Fig.5.23), however, redundancy cannot be currently automatically checked as the corresponding target constraint cannot be formulated with our limited constraint language. The required target constraint, expressed informally, is namely:

There exist no two commands without incoming next links.

This can be formalized using *nested* constraints [51], which are yet to be integrated into the precedence-driven synchronization algorithm as presented in this thesis. This means that redundancy of the source NAC in FirstGotoRule must be checked manually by the user using knowledge about all (formal and informal) target domain constraints.

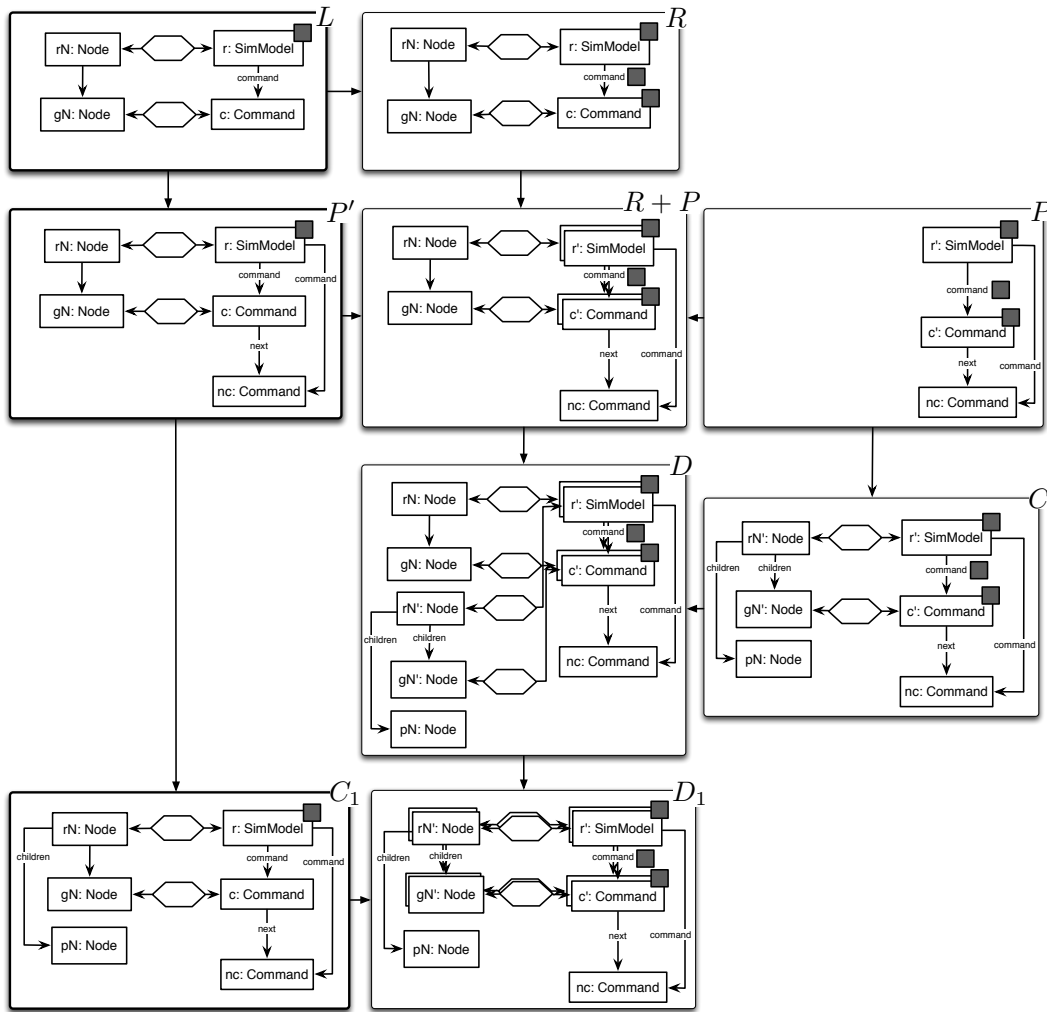


Figure 6.11: Generated non-trivial positive application condition for FirstGotoRule from the backward local completeness constraint of PaintGotoRule::V0

IMPLEMENTATION AND EVALUATION

This chapter provides an overview of the implementation of the synchronization approach presented in this thesis. Section 7.1 discusses the most important design decisions that led to the current choice of implementation technologies, while Sect. 7.2 presents a high-level overview of the core model transformations in eMoflon. The scope of the current implementation is given in Sect. 7.3, together with a qualitative and quantitative evaluation of the system in Sect. 7.4, based on the CME project and requirements R1 – R14 identified in Chap. 1.

7.1 FROM MOFLON TO EMOFLON

Moflon [1] is a metamodeling and model transformation tool with support for bidirectional model transformation via TGGs. To implement the language extensions of this thesis, the difficult decision was made in [4] to re-engineer Moflon and re-build most basic components completely from scratch. This resulted in the new tool eMoflon.

The main reasons for this decision and the corresponding design decisions that shaped eMoflon are briefly discussed in the following. The interested reader is referred to [4] for further details.

NEW MODELLING STANDARDS: When development on Moflon commenced in 2002, Java Metadata Interfaces (JMI) appeared to be the clear choice of standard Java interfaces for MOF. For several years, however, EMF Java Interfaces have now become the *de facto* standard mapping for *Ecore*, which is isomorphic to Essential MOF (EMOF), a subset of MOF.

To simplify an exchange of (meta)models with other tools, and to enable the usage of EMF-based facilities (model editor, persistency framework, etc.), the switch from JMI to EMF was an important factor in favour of an EMF compatible reengineering of most components.

MATURE SUPPORTING TECHNOLOGY: In addition to establishing a new standard, EMF also provides a standard *implementation* of the defined mapping of *Ecore* to Java. This default EMF code generator is used by a substantial number of people and is relatively stable.

The Eclipse plugin ecosystem, with its *update manager* and plugin architecture, has also established itself as a stable and much used platform for building and deploying tools in general.

Finally, a series of professional, off-the-shelf Computer Aided Software Engineering (CASE)-tools now provide extensible editors for visual modelling. An affordable and well-known example is Enterprise Architect (EA) by Sparx Systems (www.sparxsystems.com).

Especially in an academic context, where interoperability, easy installation and updating, and a strong focus on *core competencies* are paramount, a reengineering of Moflon to leverage mature supporting technologies was an important requirement. eMoflon now makes use of the standard EMF code generator, is built and released as an Eclipse plugin, and provides an extension to EA for visual modelling, as well as an Eclipse editor for textual modelling. This allows the research group to focus on *model transformation with graph transformation* as a core competence.

LONGEVITY: A major challenge when developing and maintaining an academic tool such as Moflon/eMoflon is addressing *longevity*. Most (in the case of eMoflon *all*!) developers are (PhD) students, who spend at most 4–5 years working on the tool. A crucial factor that almost forced a reengineering was the fact that there were almost no automated tests for Moflon. Refactoring an existing system is impossible without a solid testsuite [36] and making the necessary changes to Moflon appeared a daunting task indeed.

To learn from previous mistakes, eMoflon now places a strong focus on *fully automated* testing and on *bootstrapping*. Bootstrapping is the practice of using a tool to develop itself and is an important impetus for developers to continuously improve functional and non-functional features of the tool. Bootstrapping eMoflon is an important success story for eMoflon and is discussed in detail in [72].

7.2 CORE MODEL TRANSFORMATIONS IN EMOFLON

In the spirit of bootstrapping, Fig. 7.1 depicts eMoflon as a landscape of model transformations. In the following, the components that realize these transformations are discussed briefly. For further details on eMoflon, the interested reader is referred to [4, 72].

- ①: eMoflon provides a visual concrete syntax, which has been used to depict all examples in this thesis. This component is realized as an EA *extension* that consists of a set of UML profiles tailoring the standard editors in EA for specifying metamodels in Ecore, unidirectional model transformations as programmed graph transformations (GraTra), and bidirectional model transformations as TGGs. This EA extension for eMoflon also implements the transformation ① in Fig. 7.1, enabling an *export* of visual specifications from EA to a simple XML tree, and an import of such XML trees containing eMoflon specifications back to EA. In accordance with the MOCA framework presented in Chap. 3, this *platform-to-tree* transformation is realized as two unidirectional transformations implemented in C#, which are kept as simple as possible, shifting most of the complexity to the ensuing *tree-to-model* transformation ③.

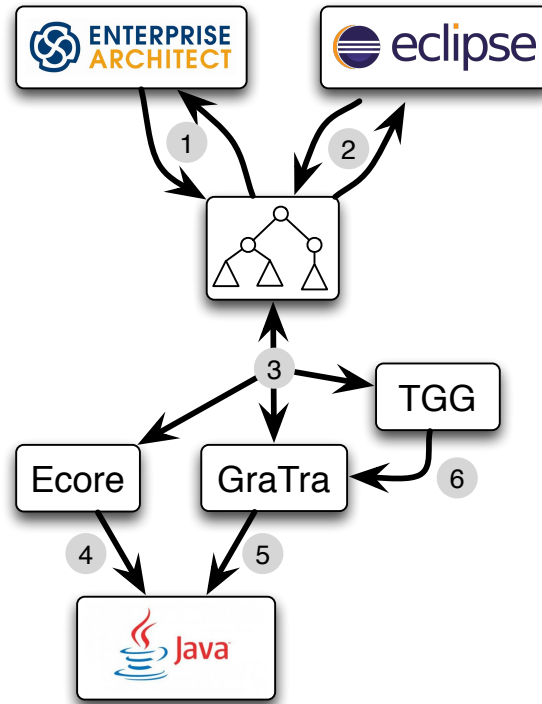


Figure 7.1: Overview of model transformations in eMoflon

- ②: For textually inclined users and to achieve a certain level of platform/tool¹ independence, eMoflon also provides a textual concrete syntax and a corresponding Eclipse text editor. Similarly to ①, transformation ② produces the same context-free tree structure from textual specifications with an ANTLR parser, and is able to “pretty print” the textual concrete syntax from such trees via a StringTemplate unparser. As MOCA provides a generic XML adapter, the parse trees² from ANTLR can be converted to and from the XML tree structure. Note that the pair of transformations ① and ② already provides a means of converting visual specifications to textual specifications and vice-versa!³
- ③: The concrete syntax tree produced by ① and ② is transformed by ③ to Ecore for metamodels, programmed graph transformations (GraTra) for unidirectional model transformations, and to TGG specifications for bidirectional model transformations. In accordance with the MDE unification principle [14], all these artefacts are models that conform to their respective metamodels. This allows for elegant higher-order transformations [14], i.e., ③ is realized as a bidirectional model transformation with TGGs⁴ that transforms Ecore, GraTra specifications, and of course TGGs as well.

¹ EA extensions currently only run satisfactorily on Microsoft Windows.

² With minor adjustments.

³ In reality this is slightly more complex due to loss of layout and other information (unique identifiers in EA, etc.).

⁴ Currently work in progress. The release version of eMoflon uses GraTra for this transformation.

- ④: The JET-based standard EMF code generator provides a unidirectional model-to-text transformation of Ecore models to EMF compatible Java code. This is used in eMoflon, but is extended to allow for implementations of methods either as GraTra specifications, or as handwritten *injections*, which are woven into the produced Java code during the generation process.
- ⑤: eMoflon supports unidirectional model transformations via programmed graph transformation [35],⁵ a combination of declarative graph transformation rules with a simple control flow language similar to UML activity diagrams. Such specifications are transformed to Java code (or interpreted) by a *graph pattern matching engine*. eMoflon currently uses *CodeGen2* from the Fujaba toolsuite [38], but changing to the new incremental graph pattern matching engine *Democles* [96] is work in progress.
- ⑥: The operationalization process described in Chap. 5 is implemented as part of a *TGG compiler*, which realizes ⑥ as a unidirectional model transformation from TGGs to GraTra, with GraTra. The operationalized TGG rules are embedded as patterns in GraTra rules and extended by bookkeeping patterns to keep track of context and created elements, and a final pattern that creates, fills, and returns a “match” data structure.

These extensions simplify the TGG-based synchronization algorithm presented in Chap. 5, currently implemented as a mixture of GraTra and Java code, which invokes the operational rules for determining precedence matches and updating the precedence graphs. The GraTra specification produced by the TGG compiler is finally transformed to Java reusing the standard graph pattern matching engine for ⑤.

7.3 SCOPE OF IMPLEMENTATION

The current release version of eMoflon⁶ contains a relatively stable and well-tested implementation of the following contributions of this thesis:

- A realization of the MOCA framework presented in Chap. 3 with wizards, standard adapters for XML and ANTLR, and optimized generated code for the standard tree metamodel *MocaTree*.
- Support for modelling TGGs visually and textually with *all* the new language features introduced in Chap.4, i.e., attribute conditions, dynamic conditions, and rule refinement. The latter is currently implemented as a GraTra transformation that “flattens” a TGG with rule refinements to a TGG without rule refinement, which is then passed to the TGG compiler.
- The synchronization algorithm exactly as presented in Chap. 5, i.e., extended to support all new TGG language features as compared to the version presented in [70].

⁵ The concrete dialect used in eMoflon is referred to as *Story Driven Modelling*.

⁶ 1.6.0 as of 16.09.2014.

- Finally, a number of runtime tools including a *graph viewer*, a visualization for all specifications with `dot`,⁷ and a *triple viewer* with which triple graphs can be visualized as a matrix. The interested reader is referred to the detailed⁸ tutorial-like documentation for end-users available from www.emoflon.org.

The current limitations and missing parts of the implementation include:

- The construction and static analysis techniques presented in Chap. 6 are not part of the current release of eMoflon.

Although initial plans for transforming TGG source marking rules to Henshin [11] for a CPA exist, this is still work in progress and probably requires non-trivial filtering and interpretation/presentation of the analysis results for end-users.

- Similarly, although a prototypical implementation of the constraint to application conditions transformation given in Fact. 11 exists from [24], which already handles attribute constraints, the transformation is currently only for negative constraints and NACs and must be extended to handle more expressive conditions. Scalability, filtering, and interpretation/presentation challenges also have to be adequately addressed before an end-user can profit from the results of the transformation.

⁷ www.graphviz.org

⁸ Over 250 pages as of 16.09.2014

7.4 EVALUATION

The MOCA framework and all TGG language extensions presented in this thesis have been extensively used for various student projects, industrial projects such as the CME case study, and finally, internally in eMoflon, e.g., for the import/export transformation ③. In the following, the CME case study⁹ will be used to provide an evaluation based on the requirements identified in Chap. 1.

As the actual implementation is in principle similar to the variants of the running example used in previous chapters, it is not necessary for the reader to understand each of the 46 TGG rules of the CME case study in detail. Where necessary, excerpts of TGG rule fragments or rule refinement networks are depicted and explained.

7.4.1 Text Generation, Unparsing (R1), and Text Parsing (R2)

In accordance with the MOCA framework, a clear and strict separation was enforced between the text-to-tree and tree-to-model transformations (forward direction), as well as between the model-to-tree and tree-to-text transformations (backward direction). As all complexity was shifted to the tree-to-model and model-to-tree transformations, very simple straight-forward templates (using String-Template [82]) were sufficient for text generation. For both formats, i.e., CLS and MPF, only about 30 lines of code were required.¹⁰ For parsing, a standard parser generator ANTLR [81] was successfully used to produce lexers and parsers for both formats from about 40 lines of code.

A practical advantage of keeping parsers and unparsers simple is their stability, i.e., the CAM target metamodel was iteratively adjusted and extended without having much effect on the parsers and unparsers.

7.4.2 Change Detection (R3)

Detecting changes made to the formats, i.e., *delta construction* was difficult as standard model diff libraries such as *EMF compare* produced correct but often disappointing and unexpected deltas. As the CME case study is inherently “offline” and requires delta construction from two different versions of a textual format, a specialized diff procedure had to be implemented for the fortunately relatively simple target metamodel. This guaranteed that the produced deltas corresponded to the actual changes made to the formats and were as expected. The experience here is that delta construction is difficult and should be avoided if possible (unless this is to be the primary focus of the research project). One reason is that “the” delta describing the set of changes leading from one model to the other is in general not unique. It is non-trivial to characterize precisely which of the possible deltas is to be “preferred”. Even for the small target metamodel, it was still time consuming and difficult to implement a robust delta recognition procedure.

⁹ For a quantitative evaluation of the import/export transformation ③, the interested reader is referred to [72].

¹⁰ Note, however, that only a small subset of both formats was covered.

TGGs are delta-based and require deltas, so this should be taken into account as early as possible when deciding if a synchronization task can be addressed adequately with a TGG based approach or not. It is also conceivable that users specify or correct the constructed deltas manually to achieve a high quality of synchronization results. This improvement of “delta quality” can also be automated to a certain degree using graph transformations [62, 63].

7.4.3 *Information Preservation (R4)*

The parsers and unparsers were implemented to be information preserving, meaning that the only lossy transformations were the tree-to-model and model-to-tree transformations specified with TGGs. Using the incremental synchronization algorithm, it was possible to deal adequately with unrelated information in the trees, ensuring that even all comments and layout were preserved during updates.

Certain deltas, however, disrupt the chain of linked operations and force a re-translation of, in the worst case, all previous operations. This leads to a consequent loss of previous comments, layout, and unrelated information. In general, information preservation must be evaluated on a delta-by-delta basis. Depending on the TGG, certain deltas (for the CME project, attribute changes) can be well supported, while others (for the CME project, deleting or inserting new operations in the middle of a chain) lead to a loss of information in order to guarantee correctness. This means that frequent or important deltas should be prioritized and taken into account when designing the TGG and specifying dependencies between rules. Improving incrementality (information preservation) of the synchronization is crucial future work.

7.4.4 *Bidirectionality (R5)*

In accordance with the lightweight MOCA-based approach depicted in Fig. 3.9, the realized CME synchronization setup consisted of two TGGs, one for CLS and the other for MPF. The setup was completely symmetric, i.e., changes to both formats could be handled via forward/backward incremental delta propagation.

It is, however, wrong to assume that this bidirectionality is completely free of charge. When developing a TGG, even experienced developers need to pause and “think” now and then in forward and backward directions for each rule, in order to validate if the synchronizer behaves as expected in both directions. With some training, this should nonetheless be better than maintaining two completely separated transformations.

7.4.5 *Efficiency (R6) and Scalability (R7)*

To evaluate the efficiency and scalability of the synchronization, the initial forward and backward transformations for the CLS format were measured for all involved components (parser, TGG, unparser). This is a prerequisite for synchronization and also provides a context for evaluating the attained speed-up with incrementality, i.e., assuming information loss were irrelevant, deltas *could* be propagated via

a batch transformation that simply re-translates all elements. Apart from avoiding information loss, therefore, an incremental solution is useless as soon as it is slower than the corresponding batch transformation for a given delta.

In addition, a simple attribute value change was chosen as a delta that can be theoretically propagated in constant time independent of model size. The backward propagation with the synchronization algorithm was measured for applying this delta to the exact same models, used for measuring the runtime of the batch transformations, and for propagating this change to the CLS trees.

Figure 7.2 depicts a plot of runtime (y-axis with a logarithmic scale) for CLS files ranging from 10 lines of code to 10,000 lines of code (x-axis with a linear scale). In each case, approximate tree size in number of nodes is given by multiplying lines of code by a constant factor of 7, approximate model size by a constant factor of 4. For example, the largest tree and model consisted of about 70,000 ($7 \times 10,000$), and 40,000 ($4 \times 10,000$) nodes,¹¹ respectively.

The x-axis is divided into three intervals with varying step sizes: 10 – 100 with step size 10, 100 – 1,000 with step size 100, and 1,000 – 10,000 with step size 1,000. This means that the gradient of the curves must be regarded and interpreted separately in each interval. Each data point was repeated 11 times with the median shown in the plot in order to ignore outliers mainly due to garbage collection and the “cold start” of the Java virtual machine. A standard PC was used for all measurements with 16GB RAM, an Intel Core i5-3550 CPU @ 3,30 GHz, 64 Bit Windows 7, Java 1.8, and Eclipse Luna (modelling version).

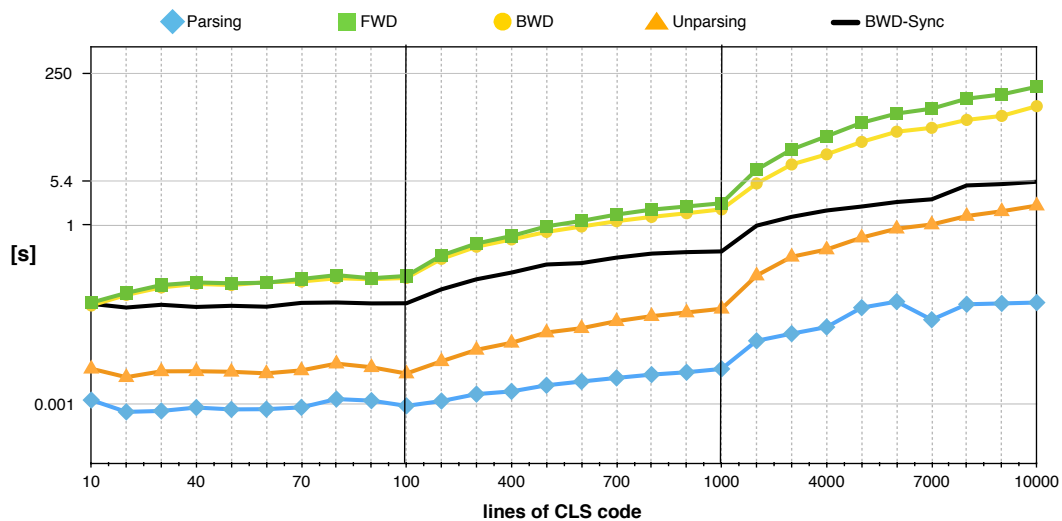


Figure 7.2: Runtime of batch transformations and backward synchronization

The runtime for *parsing* (diamond-dotted blue line), for the TGG *forward batch transformation* (square-dotted green line), for the TGG *backward batch transformation* (circle-dotted yellow line), for *unparsing* (triangle-dotted orange line), and for the TGG *backward synchronization* (solid black line) of the same delta in all cases, was measured using the `System.nanoTime()` function provided by Java.

¹¹ With approximately the same number of edges.

The test data was generated by repeatedly appending a small example from the industrial partner as often as necessary to attain the required file size. While certainly synthetic, the thus generated files are not completely unrealistic as real-world CLS files do have such repeated and only slightly changed sections. This evaluation strategy was discussed with the industrial partner prior to running the measurements and was preferred to “real” CLS files as it was difficult to restrict the existing CLS code generator used by the industrial partner to exactly the subset of the language supported by the TGG specified for the CME project.

In the domain of manufacturing engineering, it is quite difficult to state the “typical” file size or lines of CLS code. This depends on the geometry of the manufactured object, and there do exist realistic files in the range of 10 - 10,000, but also substantially larger files. The measurements depicted in Fig. 7.2 were stopped at 10,000 lines of code due to memory problems.

The following can be observed from the measurements:

- The bottleneck of the transformation chain is clearly the TGG transformation. In this special case, this is not surprising as the textual formats are very simple, it is, however, also clear that string parsing algorithms are mature and quite efficient for context-free parsing [81]. The conclusion here is that the runtime scalability of TGGs is relevant and crucial for such transformation chains.
- The TGG forward and backward transformations do not appear to be exponential as the curves (in each interval) are not straight lines.
- The forward and backward transformations are fairly symmetric. This is remarkable as the source patterns in the TGG rules are much larger than the target patterns. The cost of pattern matching should, therefore, be substantially greater in the forward direction. Furthermore, models are about half the size of the corresponding trees (factor 4 vs. factor $7 \times$ lines of code). Finally, models are ideally connected structures, while CLS trees are context-free and require numerous attribute and dynamic conditions to identify implicit connections.

The reason why the forward transformation is almost as fast as the backward transformation (actually faster in some cases considering the difference in size), is because the generic MocaTree metamodel has been hand-optimized (cf. Chap. 3). This is one of the advantages of using such a standardized and generic tree metamodel as a fixed interface for XML, parse trees, etc.

- Concerning the backward synchronization (solid black line), the measurements show that the current implementation is *not* strictly efficient as defined in Def. 20. From a practical point of view, however, incrementality still makes sense even if a certain dependence on model size is present. For the largest model, backward synchronization takes 5.4s while the batch backward transformation takes about 100s. The curve for backward synchronization is also more of a “stair-case”, i.e., with efficient intervals and jumps in-between.

Further insights gleaned from profiling include:

- The current primary hotspots of the transformation and synchronization are not yet pattern matching as expected, but are rather standard operations on EMF related data structures (e.g., `EList.add`), which simply do not scale as expected. This means that, at least from a scalability perspective, the synchronization algorithm¹² should be completely re-implemented in Java to avoid EMF data structures as much as possible. A further reason for the current dependence of backward synchronization on model size (0.05s – 5.4s) is an explicit handling of edge wrappers as EMF does not treat edges as first class objects. Attaining a “near efficient” implementation by avoiding EMF data structures and establishing a better bookkeeping of edge wrappers is future work. In any case, no fundamental problems with the synchronization algorithm such as a bottleneck when collecting matches could be identified with the case study. The lesson learned here is that EMF/GraTra is (currently) better suited for static analyses and for the TGG compiler (operationalization) as these tasks do not have to deal with very large models.
- The current integration of attribute conditions and dynamic conditions in the graph pattern matching engine is naïve and must be improved so that better “condition aware” search plans¹³ can be generated. User-defined costs or hints could also be extremely helpful. This is all work in progress and is part of the development of Democles [96], a new graph pattern matching engine for eMoflon.
- Problems with memory scalability are similarly partly due to EMF data structures and are further aggravated because of the numerous auxiliary data structures required for the synchronization algorithm such as both precedence graphs (containing all potential matches!), the translation protocol, and the correspondence model. The current focus is, however, more on runtime efficiency than on memory efficiency although this should also be shifted in the near future.

7.4.6 Configuration and User Interaction (R8)

The CME TGGs were designed to incorporate runtime configuration and decisions into the synchronization process. A configuration component can thus be used to decide, e.g., how “modal” the tree and generated code should be. This is fixed in the forward direction but is a freedom of choice during backward synchronization for the MPF format (cf. Chap. 1). This was possible without any extra effort and without risking an incorrect or incomplete synchronization as the theory established in Chap. 5 embraces non-confluent TGGs [8]. In discussions with our industrial partner, accommodating such runtime design decisions is a crucial feature as, at least in the context of the CME project, developing and maintaining a separate TGG for each possibility was not a feasible alternative.

¹² The control algorithm and not the TGG compiler used to operationalize TGG rules!

¹³ The order in which the elements of a match are determined.

7.4.7 Conflict Detection and Resolution (R9)

As discussed in Chap. 1, conflict detection and resolution is out-of-scope for this thesis and was not investigated in the CME project. It is, however, highly relevant as concurrent changes to both formats *are* made in practice, inevitably resulting in conflicts.

7.4.8 Expressiveness (R10)

Of the 46 TGG rules specified for the CME synchronization chain, 5 rules were *ignore rules* used to formalize information loss. There were many more ignore rules (almost half at one point) during the first iterations of the project. These were, however, iteratively refined and finally removed as more of the syntax was covered by the corresponding TGG. The lesson learned here is that ignore rules are a lot of work to specify at the beginning of a project, but help formalize the current state of relevant and irrelevant information.

Every TGG rule (apart from some ignore rules) contained at least one attribute condition. In most cases two to three attribute conditions were the norm. In addition to a set of built-in attribute conditions for string manipulation and basic arithmetic, some user-defined transformation-specific attribute conditions were also implemented to realize a relaxed form of equivalence of coordinate values, which can be the same up to different precisions of the format and, in some advanced cases, even up to an application of a rotation matrix. It was very useful to have the full power of Java for arbitrary attribute manipulation via the clean and simple interface of attribute conditions.

To search for certain paths in the parse trees, 11 source dynamic conditions were implemented as graph transformations, and later replaced during an optimization phase with Java code to exploit caching of relevant values. This was a helpful transition from a more to a less declarative specification as required for dealing with hotspots in the transformation.

As a conclusion, specifying the TGGs for the CME project *without* attribute and dynamic conditions would require non-trivial pre- and postprocessing of the trees, which is difficult to consolidate with deltas and the synchronization algorithm.

7.4.9 Productivity and Maintainability (R11)

There were in sum 46 TGG rules, with 17 *abstract* rules. The maximal refinement depth was 3, with a total number of 42 refinement relations, and a maximum of 3 multi-refinements. These numbers imply that (i) the refinement networks were more flat than deep, (ii) (multi-)refinement was heavily used, and (iii) common rule fragments are usually not complete rules and are thus made abstract.

To give an impression of a refinement network used in practice, Fig. 7.3 depicts an excerpt of one of the TGG rule refinement networks used in the CME project. Current experience indicates that multi-refinement (fan-out) is used to compose rules from modular fragments, while the depth of the refinement network indicates adjustments akin to an overriding of information in the basis rule.

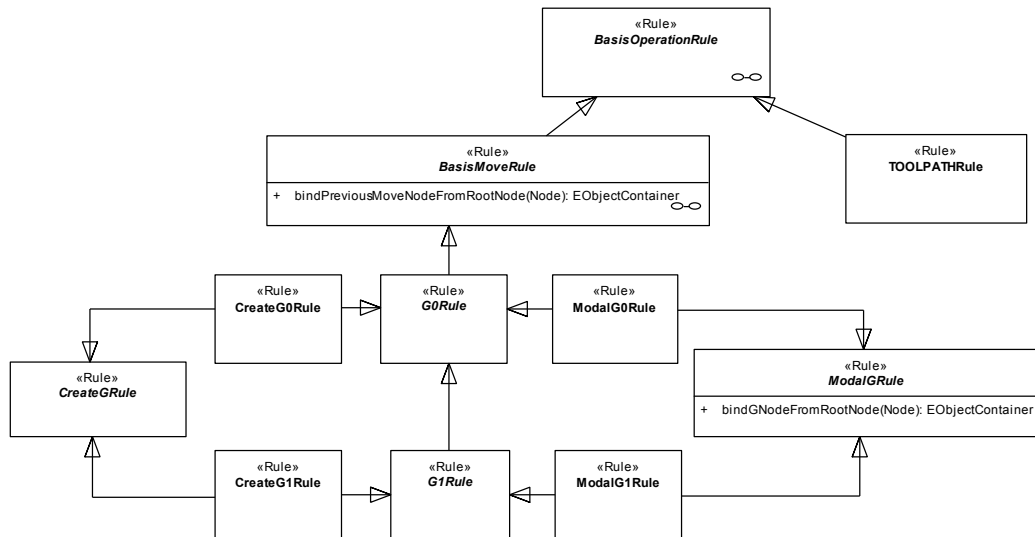


Figure 7.3: An excerpt of a TGG rule refinement network used in the CME project.

Using rule refinement leads to smaller and more focussed TGG rules. To give an impression of the average size and complexity of a single rule, Fig. 7.4 depicts an average TGG rule¹⁴ from the CME project. During reviews and when debugging, the resulting refined rules were often visualized using dot/graphviz to give an overview of the final rule with all its elements.

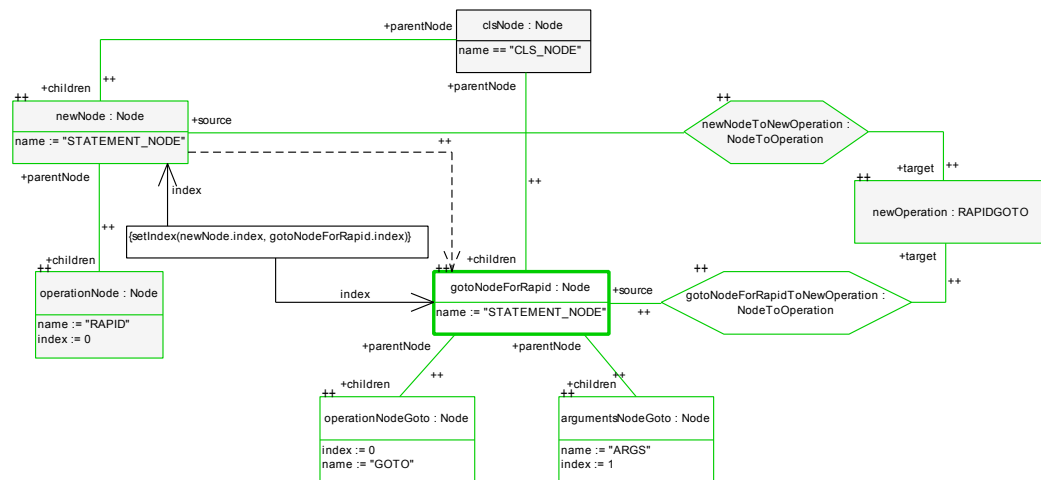


Figure 7.4: A sample TGG rule of average size and complexity.

The experience from the CME case study indicates that rule refinement greatly helps in structuring, maintaining, and refactoring TGG specifications. From iteration to iteration, the number of rules increased from about 10 to 140, and then reduced, after massive refactoring, parser optimization and extraction of common parts, to the current 46 rules.

¹⁴ Recall that all grey nodes are refined from basis rules.

7.4.10 *Validation and Static Analysis (R12)*

The TGGs were manually validated via multiple reviews and a testsuite. A synchronization editor was also built as a prototypical “end product”.¹⁵ Automating these checks as discussed in Chap. 5 is important future work, especially for non-experts.

7.4.11 *Framework (R13) and Established Standards/Conventions (R15)*

Having an established practical framework such as the MOCA framework from Chap. 3 was very helpful when organizing and implementing all components for the case study. Especially the multiple delta construction, model diff, and propagation steps depicted in Fig. 3.9 can be confusing without a schematic overview and an established way of doing things.

The maintainability of different instances of the same framework (e.g., the internal import/export module for eMoflon) is also improved as the overall structure is repeated and can be easily recognized.

7.4.12 *Theoretical Foundation (R14)*

The implementation of the synchronization algorithm was greatly simplified by having a solid theoretical foundation. Although the static analyses have not yet been automated, it was nonetheless extremely helpful to have clear assumptions and properties to check against, when understanding why certain tests did not produce expected results.

¹⁵ The interested reader is referred to a demo video available from www.emoflon.org presenting this editor as part of the practical results of the CME project.

RELATED WORK

This chapter gives an overview of approaches and state-of-the-art results that are related to the contributions of this thesis. The goal is to provide a context in which to view the results of this thesis, and to discuss strengths and limitations of alternative approaches. The focus is placed on general *approaches* and not on concrete *tools* that usually implement a combination of ideas. When discussing a particular approach, however, suitable representative implementations are of course mentioned.

The different groups of related approaches are structured according to the different contributions of this thesis:

SECTION 8.1:

Alternative approaches to organizing a landscape of components for synchronization are discussed in Sect. 8.1, providing a comparison to the MOCA framework presented in Chap. 3 in each case.

SECTION 8.2:

The choice of TGGs as a bidirectional language is discussed in Sect. 8.2, providing a broad classification of bidirectional approaches in general.

SECTION 8.3:

Related approaches concerning the TGG language extensions presented in this thesis (attribute conditions, dynamic conditions, and modularity concepts) are presented in Sect. 8.3.

SECTION 8.4:

Alternative TGG-based synchronization algorithms and differences to the precedence-based algorithm presented in this thesis are discussed in Sect. 8.4.

SECTION 8.5:

Finally, Sect. 8.5 gives an overview of restrictions that other TGG-based approaches require, and corresponding static analyses (if such exist).

8.1 FRAMEWORKS FOR MODEL SYNCHRONIZATION

A framework for organizing and structuring all components necessary for a synchronization chain was identified as an important requirement in Chap. 1.

The choice of such a framework must be evaluated based on the other requirements identified in Chap. 1, especially *Information Preservation (R4)* and *Bidirectionality (R5)*. In the following discussion, based on [5], different general strategies for organizing a synchronization chain are presented.

8.1.1 *Combination of Unidirectional Approaches*

Although there is an increasing number of bidirectional languages available, the standard way of implementing bidirectional transformations is still to combine two unidirectional transformation languages, one for each direction.

Typical combinations include parser generators for text-to-model transformations such as *ANTLR* [81], and template languages such as *Xpand*,¹ or *Velocity*² for model-to-text transformations. For the ensuing bidirectional model-to-model transformations in the chain, a combination of languages can be chosen from the numerous unidirectional model transformation languages available, depending on requirements. The reader is referred to, e.g., [21, 76] for a detailed overview of mainly unidirectional model transformation languages.

The primary advantage here is clear: a combination of standard, mature unidirectional approaches is very general and “gets the job done”, while existing bidirectional approaches are mostly still in development and are often not usable for real-world application scenarios, although they might work very well for a restrained class of problems. Similarly, standard unidirectional approaches typically scale well with respect to runtime and memory consumption.

A challenge, however, is handling *incremental* changes, i.e., coping with information loss, as this becomes difficult when separate languages are used, one for each direction. Scalability also becomes a major problem if the scenario involves many small changes applied to large models.

A further disadvantage of a combination of unidirectional approaches is that it is hard to maintain: changes to the forward transformation have to be carefully reflected in the backward transformation and vice-versa, and this gets increasingly difficult with the complexity of the transformation. Productivity also suffers as two separate specifications have to be implemented. A bidirectional language is advantageous in both cases [22].

8.1.2 *String Grammar-Based Approaches*

String grammar-based approaches are centred around a single string grammar as a generative specification of the textual format to be handled. As depicted in Fig. 8.1, the main idea is to derive as much as possible from the string grammar, i.e., not only a parser, but also a metamodel (or some other form of abstraction), an editor,

¹ OpenArchitectureWare, <http://www.eclipse.org/gmt/oaw/>

² The Apache Jakarta Project, <http://jakarta.apache.org/velocity/>

and an unparser. Prominent examples of string grammar-based approaches include *Xtext* [27], *Spoofax* [60], and *Monticore* [48]. The interested reader is referred to [44] for a detailed survey of mostly string grammar-based approaches.

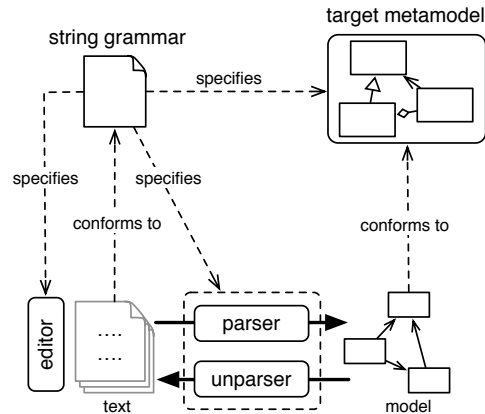


Figure 8.1: Schematic overview of string grammar-based approaches

A metamodel can be extracted from the string grammar either via an implicit transformation from EBNF to a modelling language (Ecore in the case of *Xtext*), or by extending EBNF to a complete modelling language which can be used to specify both the textual concrete syntax *and* the abstract syntax of the language combined in the grammar. The latter approach is taken by *Monticore*.

Bidirectionality can be supported by using the non-terminals in the grammar to *pretty print* model elements to text.

As pure EBNF can only describe trees (context-free structures), most string grammar-based approaches provide extensions to EBNF to allow context-sensitive relationships, e.g., via a user-defined *resolution* process in *Xtext*.

String grammar-based approaches lead to compact, concise specifications and are highly productive when the target language can be described with the string grammar. Getting an editor “for free” is also a major productivity boost, especially when developing a textual DSL.

In general, however, every string grammar dialect can only describe a limited class of languages, and, due to the fact that the string grammar is used to derive all other components, a fall-back to Java similar to what parser generators such as ANTLR offer cannot be trivially supported. Realistic transformations, therefore, typically require a subsequent model-to-model transformation, especially when the target metamodel was established *before* the textual syntax. In many cases, e.g., round-tripping as opposed to DSL development, the textual syntax *and* the target metamodel are fixed and already exist, making it challenging to specify a perfectly fitting string grammar retrospectively.

Supporting bidirectionality is also difficult in complex cases and most approaches do not place a strong focus on bidirectionality, only providing a default pretty printer that must be extended and refined as required. Even if the approach supports bidirectionality, e.g., *Xtext*, the focus is more on establishing a concrete textual syntax, i.e., not necessarily dealing with information loss. Most string grammar-based approaches are not incremental or delta-based, and do not provide

explicit support for synchronization. This might not be a major problem if the goal is to establish a textual DSL with an editor/pretty printer, but if the goal is to traverse abstraction levels as in this thesis, a string grammar-based approach must typically be combined with other approaches that can cope with information loss.

Finally, the price of having a compact, concise specification is that all components are merged making it difficult, if not impossible, to reuse the text comprehension part of the string grammar for a different target metamodel, or to change the textual syntax but retain the same metamodel. In other words, string grammar-based approaches tend to violate the principle of *separation of concerns* if they are not used solely to establish a textual front-end to a model, i.e., as a first restricted step (without information loss!) in a model synchronization chain.

8.1.3 Template-Based Approaches

Template-based approaches such as *Xround* [20] and the template-based reverse engineering approach of [19] provide an interesting contrast to string grammar-based approaches by deriving the complete bidirectional transformation from a set of templates. As depicted in Fig. 8.2, a set of templates in a fixed template language is used to derive an unparser (code is simply generated with the templates) *and* a parser. The parser works by matching text fragments with potential templates until the exact sequence of chosen templates can be identified. Corresponding model elements can be derived from this sequence of templates as the target metamodel is fixed and known to the parser, i.e., there is a mapping between model elements and templates.

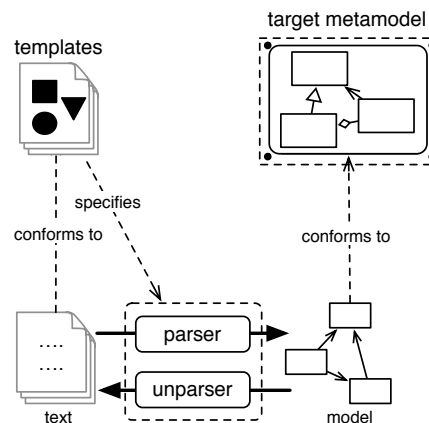


Figure 8.2: Schematic overview of template-based approaches

In contrast to a string grammar-based approach, this works well for cases with large parts of static text which must be ignored/generated. For a typical textual DSL, however, with almost a 1-1 relationship between text and model elements, the templates must contain a lot of logic and not so much static text, reducing readability and maintainability of the templates.

Although the generated textual syntax can be flexibly varied, the parser can only be realized efficiently if the template language *and* the target metamodel

are fixed. This means that a template-based approach is a productive, maintainable solution for a *fixed metamodel*, i.e., a concrete application. The parser must, therefore, be probably adjusted for every new target metamodel and cannot be completely realized as a generic component.

Depending on the complexity of the supported template language, it can also be challenging to parse textual content using templates in a scalable manner. On the one hand, complex logic in the templates can easily lead to an explosion of the template search space, while on the other hand, a parser generator based on a very simple template language will also be limited in expressiveness.

Template-based approaches have similar strengths and weaknesses as string grammar-based approaches: conciseness and good maintainability as only a single specification is used, but no support for incrementality and an inadequate separation of concerns. Template-based approaches are thus best suited for reverse engineering (sub)tasks and not necessarily for model synchronization.

8.1.4 Projectional Approaches

A further group of approaches are tightly integrated software development environments that provide *view-based*, *syntax directed* / *projectional* editing, keeping the concrete and abstract syntax of models synchronized at all times. An editor is provided, which operates directly on the abstract syntax of a model and reflects all changes immediately in the presented concrete syntax (the view). Examples for such environments include *MPS*³ and *Ipsen* [77].

The important point here is that there *is no text*, only a projected presentation of the abstract syntax that *appears* to be textual in nature.⁴ This is a powerful approach as it easily supports switching between different languages and different presentations (textual, tabular, and even visual). A reuse and composition of existing language specifications is also elegantly supported.

A syntax directed editing approach usually has rich support from the corresponding framework/environment with which the transformation can easily be specified, i.e., although this depends on the concrete environment, the process is usually productive and the resulting transformation is maintainable as it is bidirectional. Scalability, especially with respect to memory consumption, is challenging and depends on the concrete environment, but *incrementality* can easily be supported with such a tightly integrated approach.

A major disadvantage is a high dependency on the enclosing framework. This becomes problematic when the transformation is to be ported to a new modelling standard or a component has to be replaced. A further disadvantage is that an on-the-fly synchronization of concrete and abstract syntax might not be possible in some application scenarios, as text files might have to be changed “offline” such as in the CME project investigated in this thesis. Furthermore, most approaches in this group are geared towards DSL development and are not suitable for scenarios where large parts of static text must be generated.

³ JetBrains, Meta Programming System, <http://jetbrains.com/mps>

⁴ Ipsen additionally provides a textual representation and a parser with support for incremental change propagation. MPS provides interfaces and support for building custom persistency layers.

8.1.5 Model-Based Approaches

The main idea of a model-based approach to structuring a synchronization chain is to use a bidirectional model transformation language that can handle information loss appropriately with dedicated support for incrementality.

Although bidirectional languages exist that can directly deal with text, XML, and other formats, most bidirectional languages require models that conform to a certain standard. The first step in the chain is, therefore, usually to establish a suitable first simple model (typically a tree of some kind) from the input format with minimal effort. In a second step, the chosen bidirectional language can then operate on this simple model to produce the actual target model.

The complete approach applied to synchronizing multiple formats is presented in Chap. 3 and bears some resemblance to the *horse shoe* reverse engineering approach of [61]. The MOCA framework established in this thesis, however, adds support for bidirectionality and a suitable handling of deltas.

The flexible combination of different components and the possibility of implementing parts of the transformation in standard (unidirectional) languages is a pragmatic compromise with the goal of retaining the best of both worlds, i.e., using standard and mature unidirectional transformation languages for the first step, but still profiting from the advantages of a bidirectional language, especially if the primary focus can be shifted to the second step (the tree-to-model transformation).

Concerning textual formats, the strict separation in text-to-tree and tree-to-model transformations makes it possible to combine the approach with string grammar-based, template-based, and projectional approaches as follows:

- The text-to-tree transformation is restricted to (i) produce only a tree, and (ii) avoid loss of information. String grammar-based approaches work best under exactly these two restrictions and can be used to implement this first bidirectional step in the synchronization chain.
- The restricted nature of the text-to-tree transformation leads to simple templates without much logic, and, combined with the *fixed* tree metamodel (which serves as the interface between the two steps in the framework) proposed in Chap. 3, a template-based approach would also be suitable for implementing the bidirectional text-to-tree transformation.
- Projectional approaches can be integrated in a model-based framework by using them to edit the tree directly. This means that the textual concrete syntax is simply a view (projection) of the tree and the editor reflects changes made to the tree directly in the concrete syntax. The primary advantage here is a consequent support for incrementality in all phases of the synchronization chain, from the editor to the final target model.

As the main argument for a model-based approach are the advantages offered by bidirectional languages, the choice of a suitable bidirectional language is of paramount importance. The following section discusses different groups of bidirectional languages and their relative strengths and weaknesses.

8.2 BIDIRECTIONAL TRANSFORMATION LANGUAGES

There exist multiple approaches to supporting bidirectionality with a surprisingly high-level of diversity. This is because bidirectionality is a cross-discipline requirement that cuts across various domains and communities including: the database community, the programming language community, the software engineering community, and the graph transformation community [22].

Although the last two communities are gradually growing towards each other with a common MDE vision, the database and programming language communities remain fairly disjunct. A goal of the relatively new BX community is, therefore, to bring together all these groups and promote cross-fertilization [22].

In the following, five fundamentally different approaches to supporting bidirectionality are discussed. Again the focus is more on general approaches and not on concrete implementations or formal frameworks. For the former, the interested reader is referred to [91] for a survey of the BX (tool) landscape, and for the latter, to [92] for a generic algebraic framework (together with a discussion of alternative formal frameworks) for BX.

8.2.1 BX Programming Languages

The main idea behind *bidirectional* programming languages is to establish a new computation model for *reverse computability*. In analogy to Turing completeness, reverse-Turing completeness can be characterized as in [13].

An example of such a bidirectional programming language is *Janus* [101], which provides reversible basic programming primitives. A simple example is a reversible while loop, which has not only an “entry” condition, but also a “termination” condition, and can be reversed by evaluating the while expression the other way round, i.e., the entry condition becomes the termination condition and vice-versa. Other basic control flow primitives are constructed in a similar fashion.

A different example is *GroundTram* [55], a bidirectional language that requires specifications in UnQL+, an SQL-like language based on the graph query algebra UnCAL, which places strong emphasis on supporting compositionality. GroundTram operates on graph-like structures and aims to support model transformations in an MDE context.

Both Janus and GroundTram provide a restricted (programming) language, which a user can use to specify the *forward* direction from which a consistent *backward* transformation is automatically generated (or induced by changing the evaluation strategy).

It might be (arguably) intuitive for a user to specify (and think) only in one (forward) direction, relying on the underlying “compiler” to produce the other (backward) direction. Bidirectional programming languages, however, typically require determinism in both directions, which is often not the case in real-world synchronization scenarios where information loss is the norm.

8.2.2 *Combinator-Based Approaches*

Inspired by *functional* programming languages, the idea behind combinator-based approaches to BX is to specify a set of basic *laws*, which define “well-behaved” bidirectional primitives. Such primitives can be implemented in any programming language, as long as they are bidirectional and obey all the prescribed laws. Relying only on these laws, a combinator-based approach provides a *combinator*, which takes two or more primitives and composes them to form a larger atomic bidirectional unit, *which is guaranteed to also obey all well-behavedness laws*. In this manner, complex BX can be supported by repeatedly composing such units.

The concrete set of laws might vary but the fundamental, and powerful idea remains the same. The most well-known incarnations of the combinator-based approach use the *lenses* framework, which is discussed and compared to other formal frameworks in [26]. A lens is basically a pair of functions **get** (forward) and **put** (backward), that are “consistent” to each other. Consistency is formalized as a series of lens laws that roughly correspond to the TGG formal properties [9]. As TGGs are inherently relational in nature, while lenses are functional, the following discussion only makes sense for TGG-based *synchronizers*, which integrate an additional component (configuration files, user-interaction, etc.) to attain functionality of the synchronizer.

1. A set of “sanity” laws from the lens framework (concerning incidence and identity preservation) are fulfilled for TGGs by exploiting the correspondence model and do not have to be demanded explicitly. These laws ensure that totally irrelevant results are clearly forbidden.
2. Correctness for TGGs corresponds to the *PutGet* law (correctness of backward propagation) in the lens framework and is also used to fulfil the compositional laws. The PutGet law intuitively demands that the result of **getting** after **putting** should be the same as the input for **put**. In other words, backward propagation is correct if a round-trip is possible, i.e., $\text{forward} \circ \text{backward}$ is the identity.
3. Completeness for TGGs is implicitly demanded in the lens framework by requiring totality for **get** and **put** on the source and view “model spaces”.
4. Further TGG properties such as efficiency are irrelevant in the abstract lens setting.

In practice, a rich library of basic lenses is crucial, as well as a language that either provides support for constructing new lenses, or can construct a **put** from a **get** or vice-versa. Examples of lens implementations include *Boomerang* [18], and a Haskell library⁵ maintained by Edward A. Kmett.

It is still relatively unclear how powerful the combinator-based approach is, and the BX community is yet to characterize exactly what bidirectional transformations *cannot* be expressed as composed lenses.

⁵ <https://hackage.haskell.org/package/lens>

An important observation, however, is that the data structures that can currently be handled by lens implementations are typically simple in nature, i.e., strings, dictionaries, or trees. Arbitrary graph structures, therefore, appear to be challenging to handle with this approach.

8.2.3 Grammar-Based Approaches

The central idea of grammar-based approaches to BX is twofold: (i) provide a high-level language with which consistent pairs of source and target “models” can be generated, i.e., with which a *pair grammar* can be specified, and (ii) automatically generate synchronizers for forward and backward propagation (and potentially more) from this specification.

An advantage already of this basic idea is that both directions can be handled symmetrically, without one direction becoming dominant. This idea is applied to couple a string grammar and a graph grammar by [84], introducing the concept of pair grammars. This is extended further in [88] to handle the more general case of coupled graph grammars. The correspondence between “source” and “target” graphs is also made explicit with a third graph of traceability links leading to the concept of *triple* graph grammars. The automatic generation of synchronizers via an operationalization process and a control algorithm is also presented in [88].

Work on TGGs has inspired QVT,⁶ currently the only standard for bidirectional transformations. QVT is discussed critically⁷ in [93], and compared to TGGs in detail by [45] showing many parallels but also that QVT deviates substantially from the TGG formal framework mainly concerning semantics. Nonetheless, many of the ideas and language extensions presented in this thesis could be transferred to the QVT standard.

TGGs and QVT are currently mainly used in an MDE context and provide adequate support for graph-like structures, i.e., models. In general, grammar-based approaches have a potential that goes beyond synchronization; as a high-level specification of consistency, the grammar can be used in different application scenarios including model generation and a retrospective creation of traceability links. An application to quality assurance and test generation is discussed in [56, 99].

Challenges include expressiveness, i.e., certain consistency relations are hard if not impossible to express as a set of TGG rules. The exact limitations of grammar-based approaches are, however, still to be precisely formulated. To be fair to other approaches, the adequate handling of graph-like structures by (graph) grammar-based approaches is currently still coupled with a rather awkward handling of primitive data types, used mainly for attribution in graphs, such as strings, numbers, dictionaries, and maps.

⁶ Only the QVT-R (for relational) standard is actually relevant here, and is referred to in the following simply as QVT.

⁷ A major problem is that not all details are sufficiently formalized leading to different implementations that conform to the QVT standard but produce different results for even rather simple examples.

8.2.4 Constraint-Based Approaches

Constraint-based approaches address BX by demanding a consistency specification in form of a set of constraints. These constraints are direction-agnostic and state declaratively what conditions must hold for consistency. Synchronization is viewed as *consistency restoration*, and generic *model checkers* are used to search for solutions to “violations” of the consistency constraints.

The tool *Echo* [75] provides a QVT-like⁸ concrete syntax and transforms these specifications to a set of constraints for an underlying model checker.

JTL [34] also provides a QVT-like concrete syntax, and is based on *Answer Set Programming* (ASP), which is a form of declarative programming using the stable model (answer set) semantics of logic programming. JTL is presented and compared to TGGs in [34].

Pamomo [50] is an interesting TGG-inspired tool that uses triple graph *patterns*, similar to TGG rules, but interpreted as positive or negative constraints and not as rules to be applied to generate the language of consistent triples. This might seem to be a rather subtle shift but is actually profound: when specifying consistency using TGG rules, only triples that can be generated with the rules are considered to be consistent. When using triple patterns, *all* triples that do not violate any of the (constraint) patterns are consistent. With respect to language definition, TGGs provide a *bottom-up* strategy, where the language is created starting from the empty language (no rules) and covering more triples as rules are added. Language definition with Pamomo is in contrast *top-down*, starting with all triples (no constraints) and gradually excluding triples as constraint patterns are added. Analogously to TGGs, the formalization of Pamomo is based on algebraic graph transformations, and there are proofs for the same concepts of correctness and completeness as for TGGs.

XLinkit [78] is a constraint-based bidirectional language geared towards link creation in an XML-based context, leveraging standard Internet technologies such as XML, XPath, and XLink. XLinkit is based on a novel semantics for first-order logic that produces links instead of truth values [78].

Constraint-based approaches to BX have a considerable potential; advanced tasks such as conflict detection and resolution, characterization of least-change, and tolerating inconsistencies while synchronizing can all be handled elegantly and uniformly. The concept of a “constraint” also unifies primitive data types for attribution, complex graph-like structures, and even user-defined constraints possibly implemented with an arbitrary programming language.

The main challenge of constraint-based approaches is scalability and runtime efficiency. This is to be expected as the powerful but generic model checkers used in most cases are not aware of any graph structure and require every “connection” to be expressed as a constraint. This leads to an explosion of the search space and consequently of the effort required to solve the resulting constraint problems.

⁸ The QVT standard can also be viewed as a constraint-based approach.

8.2.5 Query/View-Based Approaches

A *query* can be viewed as the fundamental unit of data movement or transformation. Even if the actual transformation is specified at a higher level, this can still be compiled and converted into a set of basic queries.

An overview of database research into BX is provided in [22]; goals include (i) identifying under what conditions existing queries in SQL, Datalog, or XQuery can be “reversed”, and (ii) studying the *view update problem*, i.e., intercepting updates to a “view” defined as a set of queries on a “source”, and converting these view-updates to updates applied directly to the source. This must be done in such a way that re-running the queries regenerates exactly the updated view. The vision here is to identify semantic or syntactic constraints on the set of queries that determine if the defined view is updatable or not. There exist various BX-related application scenarios particular to database research [22] and with unique sets of requirements and challenges including: *data exchange*, *cross metamodel*⁹ *mapping*, and *co-evolution* of database schemas and database instances.

The need for views in an MDE context has led to an Eclipse project *EMF Facet*,¹⁰ which allows the specification of queries on models using Java, OCL, ATL, JXPath, etc. EMF Facet aims to be a lightweight, flexible, and pragmatic approach to extending Ecore metamodels with extra attributes and relations defined as a set of queries. These extensions or “facets” are, however, read-only as the current focus is not on a high-level specification of consistency and synchronizers.

8.2.6 Main Advantages of TGGs as a BX Language

To conclude this section on BX approaches, the main arguments for choosing TGGs as a basis for implementing the MOCA framework are presented in the following.

1. There exist multiple, actively developed TGG tools. A detailed qualitative and quantitative comparison of the batch and incremental capabilities of current¹¹ TGG tools is given in [57], and in [74], respectively. Although it is not yet possible to exchange TGG specifications amongst TGG tools, this is work in progress and would help to bring the already vibrant TGG community closer together.
2. TGGs are based on the solid formal foundation of algebraic graph transformations. This means that the considerable know-how of handling graph-like structures can be leveraged for TGGs. This makes TGGs particularly suitable for applications in an MDE context.
3. TGGs are general, i.e., cover many application scenarios, and implementations are still relatively efficient, at least when compared to constraint-based approaches. This is because TGG-based synchronizers are typically

⁹ Note that the term “metamodel” has a different meaning in a database context than in MDE!

¹⁰ <http://www.eclipse.org/facet/>

¹¹ As of early 2014. The TGG tool landscape is constantly changing.

implemented with graph pattern matching engines and not general model checkers, leveraging the considerable *practical* know-how of building efficient graph pattern matchers.

4. TGGs are appropriately restricted to be amenable to static analyses and construction techniques. Although this involves a careful compromise with expressiveness, it is one of the primary advantages of a restricted DSL such as TGGs over GPLs such as Java.
5. Finally, some subjective arguments include the historically and still predominantly visual concrete syntax for TGG rules, which is (arguably!) considered to be intuitive and easy to understand,¹² especially by the visually inclined.

8.3 LANGUAGE EXTENSIONS FOR TGGs

All the TGG language extensions identified and established with this thesis are based on substantial preliminary work and existing ideas from various groups and authors. This is crucial for new language features as only ample experience can show exactly which features are relevant in practice and can/will actually be used by end-users.

Based on [7, 10], an overview of the related work that led to the current set of extensions as presented in this thesis is given in the following, together with an explanation of what exactly was improved in each case.

8.3.1 *Complex Attribute Manipulation in TGG Rules*

The requirement of supporting complex attribute manipulation in TGGs is not new and has already been identified as a major deficit of TGGs by various authors including [23, 49, 64, 68, 98].

As TGGs are aligned with the QVT standard in [68], the approach taken by [64, 68] is similar to what is described in the QVT specification. As these approaches have, however, not been sufficiently formalized, it is unclear how constraints that are more complex than simple *expressions* consisting of a single parameter (which are trivially revertible), are to be handled. This restriction is neither enforced nor checked in anyway as arbitrary OCL expressions can be used, leaving the user or tool provider to decide what “makes sense” [93]. Furthermore, although *black box operations* can be integrated with *relations* (rules) in QVT, this does not allow for the same degree of compositionality and reusability attained by combining multiple (possibly user-defined) attribute conditions for a single TGG rule (cf. Chap. 4).

In [23], an integration of TGGs with OCL is presented, also allowing arbitrary OCL expressions in TGG rules. Currently only trivially reversible (attribute assignments) are supported by the implementation.

The attribute conditions introduced in this thesis support complex expressions and composition via shared variables across conditions, and can thus be viewed as a natural and necessary generalization and *formalization* of ideas from [23, 64, 68].

¹² Note that there is a huge cognitive gap between understanding (passive) and specifying (active).

Other approaches [40, 65], require the user to specify a pair of functions or constraints for each direction, which can be implemented in Java or OCL. Although such pragmatic approaches are expressive, they go against the TGG philosophy of providing a *single* specification from which different operational rules can be derived. A further problem is that the user is responsible for guaranteeing and maintaining consistency between such pairs of functions.

With attribute conditions as introduced in this thesis, only atomic (primitive) constraints need to be implemented once and can then be reused and composed freely in a declarative manner in TGG rules. Furthermore, constraint (library) providers do not need to worry about the *correct order* in which constraints must be solved as this is determined automatically, analogously to *graph* pattern matching.

Advanced application conditions are presented formally for TGGs in [43] but a corresponding TGG-based synchronization algorithm and an implementation is left to future work. The attribute conditions as presented in this thesis are much simpler application conditions, i.e., are specified only on attributes and cannot access the graph parts of the rule, but this restricted form of conditions can be handled by the TGG synchronization algorithm presented in this thesis and has been successfully implemented using the underlying graph pattern matching engine of [96].

8.3.2 *Dynamic Conditions for TGG Rules*

The concept of dynamic conditions of this thesis is used as a generic interface to abstract from various related ideas of how to extend graph patterns including *path expressions*, method invocations to an underlying language, *transitive closures* and other (path) operators.

All these ideas are formalized, e.g., in [87], implemented e.g., in [59], but are yet to be integrated in the algebraic graph transformation framework of [28]. The goal with dynamic conditions was to investigate the effect of such extensions on TGGs and on efficient TGG-based synchronization. The decision was made to do this on a high-level, i.e., also incorporating arbitrary implementations of dynamic conditions, possibly with a GPL.

I am not aware of any comparable work on TGGs. Dynamic conditions thus have to be replaced with appropriate pre- and postprocessing before using other TGG tools. This not only complicates synchronization and delta propagation, but also, from a practical point of view, shifts the focus away from TGGs. In practice, users usually do not have the required discipline to restrict the pre- and postprocessing steps to a bare minimum, i.e., TGG specifications typically degenerate after some time to pre- and postprocessing with little in-between.

The current integration of dynamic conditions in TGG rules can and should be further improved, but already induces a natural separation of concerns in a core pattern expressed with the TGG rule, and in a set of simple auxiliary methods implemented as dynamic conditions, and abstractly represented in the rule as “magic” edges whose “existence” is determined on demand.

A final advantage of dynamic conditions is explored in [73] as an *optimization technique*; after implementing a transformation with TGGs, profiling typic-

ally identifies a few primary hot spots. Such a transformation can be optimized substantially by carefully substituting certain costly sub-patterns with a dynamic condition, essentially making a trade-off between being as explicit as possible in a declarative pattern, and abstracting away details with auxiliary (and much more efficient!) methods implemented in a suitable language. Experience with the CME project and other non-trivial TGG specifications shows that a substantial improvement in runtime can be indeed obtained without completely “destroying” the declarative nature of TGG rules.

8.3.3 Modularity Concepts for TGGs

In the following, rule refinement as introduced in this thesis (in the following just *rule refinement*) is compared to existing modularity concepts for TGGs in particular, and graph transformation in general. The interested reader is referred to [100] for a broad survey of modularity concepts for model transformation languages.

Klar et al. [66] introduce a reuse mechanism for TGGs, which avoids pattern duplication by allowing rules to *refine* a basis rule. Greenyer et al. [46] extend this idea by introducing *reusable nodes*, i.e., nodes in TGG rules that can be either created or parsed as context. Rule refinement can thus be viewed as a generalization of [46, 66], as both approaches can be simulated with the following additional extensions:

1. Rule refinement supports and formalizes *multiple* basis rules, i.e., multiple refinement, which is crucial for a flexible composition of modular TGG rules.
2. In the approach of [66], every rule can only create a single distinct correspondence type. This leads to a confusing mix of two different and orthogonal concepts: (i) support for inheritance and abstract types in the metamodels (especially the correspondence metamodel) according to [28], and (ii) refinement of TGG rules. With rule refinement, this restriction is removed completely; both reuse concepts are clearly separated and can be combined freely.
3. Rather strong restrictions are posed in [46, 66] to guarantee the property that a basis TGG rule is always applicable when its refining rules are. Rule refinement does not apply these restrictions as:
 - TGGs are usually *operationalized* to derive, e.g., forward and backward transformations. In many cases, the mentioned property is violated for TGG rules *but* still applies to these operational scenarios. These restrictions thus unnecessarily limit the applicability of rule refinement in practice and are of questionable use.
 - The approach in [66] is formulated for MOF which supports advanced modularity concepts such as inheritance on *edge types*. The *de facto* standard EMF/Ecore is much simpler in this respect and, as a consequence, requires a more flexible modularity concept for TGG rules.
4. Both approaches use some form of rule priorities to resolve ambiguities caused by conflicts between basis and refining rules. As neither approach

employs backtracking due to efficiency reasons, this can either lead to wrong decisions [46], or requires the user to constantly adjust priorities as rules are added and changed [66]. This is not necessary for rule refinements as the TGG-based synchronization algorithm of this thesis uses DEC NACs (cf. Def. 54) to detect obvious dead-ends in the transformation, with a static analysis to identify cases when this is insufficient. This enables handling a well-defined class of TGGs without backtracking or user intervention.

There are numerous modularity concepts in the mature field of graph transformation. The concept of *variable nodes* in rules [58], which can be expanded to instantiate concrete rules, leads to “template” rules and requires separate, explicit expansion rules. Compared to rule refinement, this increases flexibility but also complexity. An equally related approach from algebraic graph transformation is *amalgamation* [17], where fragments of a rule can be denoted as being allowed to be matched arbitrarily many times. In this manner, a single rule can be also expanded at runtime by matching such fragments as *often as necessary*. Rule refinement is comparable to [58], but cannot be used to simulate [17]. Introducing amalgamation to TGGs is, therefore, important future work.

8.4 TGG-BASED SYNCHRONIZATION ALGORITHMS

An overview of model synchronization algorithms is provided in [74], on which the following discussion of alternative and complementary ideas for model synchronization with TGGs is based.

8.4.1 *MoTE*

The synchronization algorithm of the TGG-based tool *MoTE*¹³ is presented in [40]. Each TGG rule in *MoTE* must create a single new correspondence, which depends on all context correspondences of the rule. All created elements in the rule must be connected to this new correspondence.

The algorithm exploits the thus induced dependency tree of correspondences to determine elements affected by a modification. Correspondences connected to modified elements are sorted in a queue and processed by applying a series of consistency-restoration strategies.

First of all, an attempt is made to restore consistency based solely on changes to attribute values. If this is impossible, then an attempt is made to *repair* the rule application by adding new elements, i.e., without reverting or applying a complete rule. If this fails as well, then the algorithm deletes all obsolete correspondence and target elements (revoking the previous rule application), and re-translates all modified elements.

Compared to the precedence-driven synchronization algorithm presented in this thesis, *MoTE* uses the correspondence graph not only for traceability but also as a kind of translation protocol. *MoTE* requires *conflict-free* TGG specifications, which means that an automated strategy for avoiding conflicts, e.g., via

¹³ www.mdelab.de/mote

DEC NACs is not applied. This greatly simplifies the synchronization process (recall that additions can only revoke rule applications due to dangling edges!), but also severely limits the class of TGGs that can be handled.

MoTE is currently the most efficient TGG-based incremental tool [74], implying that the idea of restoring consistency as locally as possible without a (costly) global sorting or strategy is effective in practice (implying that the cases where a global sorting is better are perhaps “pathological” and of little practical relevance). Definitive conclusions, however, require further investigation and experience with real-world applications.

8.4.2 TGG Interpreter

The *TGG Interpreter*¹⁴ directly interprets TGG rules without first deriving operational forward or backward rules from them.

The incremental algorithm of [47], employed by the TGG Interpreter, takes a modified graph triple, i.e., with deltas already applied, iterates over the rule applications using also a kind of translation protocol, and determines if any applications have become inconsistent. If a rule application is inconsistent, an attempt to restore consistency based on attribute manipulation is made. If this fails, then the source elements are revoked by *marking* the affected correspondence and target elements as *to be deleted*.

In a subsequent phase, all revoked source elements are re-translated. During this re-translation, correspondence and target elements previously marked for deletion are *reused* whenever possible. In a final iteration, correspondence and target elements that are still marked for deletion are finally destroyed.

In principle, the idea of having a pool of “deleted” elements, which can be reused later can potentially increase the incrementality of a synchronization algorithm. It raises, however, multiple questions including how to decide if and when a user expects an element to be reused or not. These questions are still open as formal proofs of correctness/completeness are not provided by [47].

8.4.3 Synchronization Algorithm of Hermann et al.

In contrast to the rather practical/pragmatic approaches applied by MoTE and the TGG interpreter, a formal framework for TGG-based synchronization is presented in [54]. This provides a useful conceptual framework with formal proofs for correctness of model synchronization using TGGs.

Instead of using a precedence-graph or similar structure for determining which rule applications must be revoked before the re-translation phase, the algorithm in [54] employs a complete “remarking” of the triple graph to determine the *maximal consistent sub triple graph*. This guarantees correctness without requiring the restrictions posed e.g., on NACs (schema compliance and precedence compatibility) in this thesis. Unfortunately, it remains unclear how a complete remarking as suggested by [54] can be implemented efficiently in practice.

¹⁴ www.cs.uni-paderborn.de/en/research-group/software-engineering/research/projects/tgg-interpreter.html

8.4.4 Synchronization Algorithm of Orejas et al.

Different ideas from existing TGG synchronization algorithms are incorporated in a single, efficient algorithm presented in [79]. The idea of [70] using “precedences” for efficiency is combined with the idea of reusing elements for increased incrementality from [46]. The result is a promising TGG-based synchronization algorithm, which can also be implemented efficiently.

Correctness for additions that force existing rule applications to be revoked is, however, not proven formally, and are handled by asking the end-user to decide what is to be done, or by applying possibly problem-specific heuristics.

8.5 RESTRICTIONS AND STATIC ANALYSES FOR TGGs

This section, based on [8, 6], discusses the restrictions posed by other TGG-based approaches, and the corresponding static analyses (if such exist) used to ensure that these restrictions are not violated.

Restrictions can basically be divided into two main groups: (i) how the approach restricts and handles (negative) application conditions, and (ii) what properties are demanded of the transition system generated by a TGG. A suitable choice of (i) and (ii) is typically used to guarantee that a search strategy (e.g., for forward translation) does not require backtracking and is thus in this sense efficient (has polynomial runtime with respect to model size).

8.5.1 Usage and Restrictions of (Negative) Application Conditions

In the following, three different groups of approaches that introduce and use (negative) application conditions in the context of TGGs are discussed:

NACs USED TO GUARANTEE SCHEMA COMPLIANCE:

An “on-the-fly” technique of determining a sequence of forward rules via a context-driven algorithm with polynomial runtime is presented in [89]. This algorithm supports NACs, but a proof of completeness is left to future work.

In [67], an extended algorithm is presented for “integrity preserving TGGs”, a class of TGGs that appropriately restrict the usage of NACs, so that completeness, correctness and polynomial runtime can be proven. Furthermore, the algorithm uses a “dangling edge check” as a look-ahead to guarantee the correct choice of rules, and requires that a local choice between applicable rules cannot lead to a dead-end. A static verification technique for this algorithm is presented in [6] and is used to ensure correct usage of NACs as required for integrity preserving TGGs.

The characterization of allowed NACs and the main ideas for static analyses as presented in this thesis are based on [6] by Anjorin et al., and on [67].

FILTER NACs USED TO GUARANTEE POLYNOMIAL RUNTIME:

The Decomposition and Composition Theorem of [29] is extended in [32] for NACs with a similar treatment as in [89], showing correctness and completeness for a backtracking algorithm.

The “on-the-fly” technique of determining a sequence of forward rules employed in [89] is formalized in [31] and, although concepts of parallel independence are introduced and the possibility of employing a critical pair analysis are mentioned, the presented approach is still exponential in general.

This basis provided by [31] is extended in [52, 53], and a critical pair analysis is used to enforce functional behaviour. Efficiency (polynomial runtime) is guaranteed by the construction of *filter NACs*, which cut off possible backtracking paths of the algorithm and eliminate critical pairs. If all critical pairs can be eliminated, functional behaviour can be proven and polynomial runtime guaranteed.

Functional behaviour in both directions implies that the TGG describes a bijection, which is a strong restriction concerning expressiveness. Based on experience with industrial case studies such as [85], and theoretical frameworks for BX such as [92], most “interesting” BX are *not* bijections.

GENERAL APPLICATION CONDITIONS:

In [43], a larger class of general application conditions (positive, negative, complex and nested) are introduced for TGGs, and all formal results are extended appropriately. An extension of the algorithm introduced in [53], i.e., an algorithm that is polynomial and thus of practical relevance is, however, left to future work.

An integration of OCL with TGGs and corresponding tool support is presented in [23]. It is, however, unclear exactly how and to what extent the arbitrary OCL constraints must be restricted to ensure correctness and completeness of the derived translators.

8.5.2 Confluence and Local Completeness

Schürr discusses the challenge of dealing with decision points in a TGG-based transformation process, proposing two solutions in [88]: backtracking wrong decisions, or demanding confluence. For efficiency reasons, the latter is the favoured strategy taken by all existing TGG approaches I am aware of.

Hermann et al. [53] perform a critical pair analysis of forward rules and generate filter NACs to resolve critical pairs arising from obviously misleading (backtracking) paths. All other critical pairs (conflicts) must either be manually checked to be confluent, or removed by adjusting the TGG rules as required for confluence.

More restrictively, Giese et al. [42] require confluence without filtering backtracking paths automatically. OCL constraints can be used to resolve critical pairs, but this is a manual process required for both forward and backward transformations. An algorithm to automate this process is yet to be provided by [42] and it remains unclear how such OCL constraints must be restricted for correctness.

The TGG approach taken by Greenyer and Rieke [46] does not explicitly require confluence but, in case of decision points, the approach may fail to find a valid result, i.e., completeness is not guaranteed for non-confluent TGGs.

Although confluence avoids backtracking (i.e., is used to show efficiency), solves completeness problems, and can be statically checked for TGGs (cf. [42, 46, 53]), it can also be too restrictive in practical scenarios (as in the CME project) as it forces a TGG to be a bijection (a function in both directions).

The TGG algorithm in [67] is currently the only efficient and complete approach I am aware of that embraces *non-confluent* TGGs, only requiring local completeness as defined in this thesis.

As the results in [67] do not provide any means to analyse this restriction statically, [8] fills this gap by exploiting a constraint-based formalization of the required condition, making it amenable to well-known techniques in the field of graph transformations. These results are extended to cover a restricted class of NACs, attribute conditions, and dynamic conditions in this thesis.

Alternatives to a *static* analysis include analyses based on TGG rules *and* a concrete input model triple such as the dangling edge check in [67], and checks based on a precedence structure [71]. Although such analyses must be repeated for every new input model, they allow violations of properties that are not relevant for the current input model. This can be very useful in practice where a static analysis (for all possible input models) might be too restrictive.

Finally, using tools such as Groove [39] or Henshin [11], complex properties can be checked by exploring the state space generated by applying the rules of a TGG. This allows for checking arbitrarily complex conditions but suffers from the usual problem of state space explosion.

CONCLUSION AND FUTURE WORK

This thesis addressed the challenge of supporting *model synchronization*, a crucial task for guaranteeing consistency in Model-Driven Engineering (MDE) activities, especially as part of concurrent engineering processes.

Based on a concrete and real-world application scenario in the domain of Concurrent Manufacturing Engineering (CME), a set of requirements and challenges were identified and mapped to the main contributions and results of this thesis, which are summarized in this chapter together with ideas for possible extensions and future work.

The three main challenges and corresponding contributions identified in detail in Chap. 1 are repeated and used to structure the following discussion in Sect. 9.1, Sect. 9.2, and Sect. 9.3.

9.1 A FRAMEWORK FOR MODEL SYNCHRONIZATION

CHALLENGE 1, formulated and mapped to requirements in Chap. 1, was to provide a means of combining Bidirectional Transformation (BX) languages with standard technology in a systematic, standardized fashion. This challenge was addressed in this thesis by presenting a framework for model synchronization in Chap. 3. This framework handles deltas, allows the integration of a bidirectional language, and enables a flexible combination of existing parser/unparser technology. Using the framework, the required components for realizing a complex synchronization task such as for the CME project (Chap. 1) could be organized and structured.

The *conceptual* eMoflon Code Adapter (MOCA) framework presented in Sect. 3.1 is independent of concrete technologies, tools, and even the choice of the bidirectional language, addressing CHALLENGE 1 on an implementation agnostic level.

A Triple Graph Grammar (TGG)-based realization of the MOCA framework, presented in Sect. 3.2, provides concrete instances of all required components and fixes a choice of technologies. Two suggested workflows (lightweight and elaborate) for handling deltas and incremental change propagation are discussed in detail in Sect. 3.3, based on an application of MOCA to the CME case study.

Ideas for future work include extending the conceptual framework to handle *integration* scenarios, which require conflict detection and resolution as part of the integration process. It is currently unclear which extra components are required for this, how user input and preferences are to be taken into account, and how much can, and should be automated.

A second area of future work is extending the framework to embrace *inconsistencies*, i.e., to accept the fact that in practice, related models are typically almost never perfectly consistent. Instead of enforcing strict consistency at all times, the framework should be built to support iteratively *improving* the situation by increasing consistency or doing nothing at all. The challenges involved in *tolerating inconsistencies* are posed and explained in detail in [94].

A further idea for future work is to exploit change information from (text) editor frameworks. As an example, Eclipse-based text editors provide information about exactly which “areas” of a text buffer have been changed and must be repaired. This is often exploited to enable incremental and thus scalable editor functionality such as syntax highlighting for large files. This information could be used in the MOCA framework to determine the exact sub-trees to be recreated. As the tree is context-free, this is theoretically feasible, and only requires that tree nodes contain line and position information (already the case in the current implementation). This extension could be used to avoid re-parsing and code generation of entire text files. This can potentially improve the quality of synchronization results by avoiding ambiguous and often costly non-incremental delta construction, without enforcing a radically different editing process such as required for projectional editing (cf. Sect. 8.1.4).

Another idea is to investigate if the current extremely simple MocaTree meta-model should be extended to include e.g., next links between sibling nodes, or even arbitrary cross-tree links. The idea here is to extend the basic context-free tree created by e.g., a standard parser, by constructing these extra links as derived relations. This must be carefully integrated in the synchronization framework, however, as such derived links cannot be created by the TGG, cannot be propagated as deltas, but can be demanded as context.

Finally, the current tool support could be substantially improved by providing a “MocaTree aware” visual editor that represents trees in a more natural and concise manner. Especially as the tree metamodel is fixed, the normal, generic abstract syntax for metamodels can be replaced with a concrete syntax specially designed for MocaTree, with e.g., suitable symbols for folders, files, and tree nodes. This is a simple and practical idea, but could go a long way in improving readability of TGG rules operating on MocaTrees.

9.2 LANGUAGE EXTENSIONS FOR TGGs

CHALLENGE 2 was to provide a transformation language that on the one hand, allows for high-level, productive, and maintainable specifications, and on the other hand, is expressive and scalable enough for real-world applications.

The central decision to use TGGs in this thesis is justified in Sect. 8.2.6. Primary arguments are that TGGs are declarative (high-level), bidirectional, and incremental, and are thus suitable for model synchronization.

Synchronizing models on *different* levels of abstractions, however, poses additional challenges for which missing language features for TGGs were identified and established in Chap. 4.

Attribute conditions for complex attribute manipulation, and *dynamic conditions* for flexible structural constraints in TGG rules, are both used to formalize a controlled integration of auxiliary methods, which can be implemented in a standard General Purpose Language (GPL). This is important for increasing the *expressiveness* of TGGs, and both features were extensively used in the CME project (cf. Sect. 7.4.8).

To improve the *maintainability* of large TGG specifications, a new modularity concept for TGGs, *rule refinement*, was established. Rule refinement combines and builds on existing ideas, with crucial extensions that enable applicability in practical scenarios (cf. Sect. 7.4.9).

Ideas for future work include providing standard (library) implementations of typical dynamic conditions including *transitive closures* of e.g., composite relations, configurable¹ *path expressions*, and a possibly *recursive invocation* of graph patterns. As dynamic conditions are only used as application conditions, however, they are *not* an abstraction for *amalgamation* [17], which allows for *creating* dynamic patterns as well. Introducing amalgamation to TGG-based synchronizers can, therefore, be viewed as generalizing the idea of dynamic conditions to “dynamic rules”, and is certainly important and interesting future work.

The integration of attribute and dynamic conditions in the graph pattern matching engine can be considerably improved. Currently, these extensions are solved separately and combined with the search plan for graph parts in a fixed order. With a more integrated approach, end users could annotate attribute and dynamic conditions with *costs* and thereby influence how these additional conditions are “mixed-in” with the normal graph parts of a rule. This could potentially enable a more efficient attribute and dynamic condition “aware” search plan generation.

In general, a precise characterization of the class of consistency relations TGGs can currently describe is necessary to identify (and justify) the need for any further language extensions.

Concerning rule refinement, possible future work includes extending refinements to include whole TGGs, i.e., adding refinement primitives for merging TGGs, removing, adding, and mixing TGG rules as required.

Establishing indications for the readability of TGG specifications is a related task, as this could be used to induce *refactorings* based on rule refinements, which can improve TGG specifications with respect to the specific readability metric.

As rule refinements can, however, also be misused to produce complex and confusing rule networks, a characterization of “good” and “bad” refinement patterns is equally important.

¹ For example via regular expressions over sequences of edge types.

9.3 A TGG-BASED SYNCHRONIZATION ALGORITHM AND STATIC ANALYSES

CHALLENGE 3 was to provide a constructive, coherent, formal underpinning of all concepts and language constructs, which can serve as a guide for a corresponding implementation.

To address this challenge, the precedence-driven algorithm of [70] was extended to handle edges, Negative Application Conditions (NACs), attribute conditions, and dynamic conditions in Chap. 6. The extended synchronization algorithm is *correct*, i.e., produces results that conform to the specified TGG, and *complete*, i.e., does not abort with an error for *local complete* (source local complete and forward local complete) TGGs. This algorithm has been implemented in eMoflon, and was evaluated in Sect. 7.4.5 with the CME project with respect to (practical) efficiency and scalability.

Chapter 5 presented ideas for a static analysis of all properties required for correctness and completeness of the synchronization algorithm, including:

- A construction technique to produce precedence-compatible and schema-compliant NACs from negative constraints.
- A static analysis of source local completeness based on a Critical Pair Analysis (CPA) of *source marking rules*.
- A static analysis of forward local completeness based on induced *forward local completeness constraints*, which are transformed to application conditions to check for possible violations with TGG rules.

As future work, the current synchronization algorithm could be improved by applying a more detailed analysis of exactly when rule applications must be revoked. For example, as Lemma 4 shows that additions can only invalidate existing rule applications due to dangling edges, a static analysis of the TGG can be used to determine exactly which cases can lead to dangling edges. For instance, if a TGG does not require any Dangling Edge Check (DEC) NACs to avoid dangling edges, then additions can *never* invalidate existing rule applications and the algorithm can be optimized accordingly. This would improve the incrementality of the synchronization.

Based on profiling results and a hot-spot analysis for the CME project, the runtime of the control algorithm could be substantially improved by switching from standard Eclipse Modeling Framework (EMF) data structures to more suitable data structures for maintaining large sets of (partial) matches and dependencies between matches. Furthermore, certain phases of the algorithm, e.g., collecting all matches (with COLLECTDERIVATIONS, cf. Sect. 5.2.1), can be *parallelized* to take advantage of multiple cores. The actual speed-up that can be achieved with this must, however, be thoroughly evaluated.

Further extensions to the algorithm include handling of *attribute changes*, which are currently translated into deletions and additions. Handling attribute changes directly would greatly improve the incrementality of the algorithm.

The idea of inducing local “repair rules” from TGG rules [41] is a further means of improving incrementality and is interesting but challenging related work. It

is currently unclear how correctness can be guaranteed in the presence of such repair rules, which are only fragments of the actual TGG rules.

From an end-user's perspective, it would also be reasonable to support backtracking as a fallback solution. Currently, the synchronization algorithm will potentially fail for TGGs that are not local complete. It would be better to revert to backtracking, i.e., to accept and handle all possible TGGs but warn the end-user that the TGG is not local complete and could thus result in a very inefficient synchronizer.

Finally, tool support for TGGs is still in its infancy. Especially a dedicated *debugger* is sorely missing, with appropriate concepts of "stack frames", breakpoints, and filtered intermediate results, all tailored suitably to the precedence-driven TGG-based synchronization algorithm.

Concerning the ideas for static analyses presented in Chap. 5, future work includes an efficient, scalable implementation, as well as an extension of the theory (construction technique, CPA) to cover graphs, which integrate attribute conditions as part of their basis graph model and not only as application conditions.

BIBLIOGRAPHY

- [1] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture - Foundations and Applications, Second European Conference, ECM-DA-FA 2006*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375. Springer, 2006. (Cited on Page 209.)
- [2] Anthony Anjorin and Marius Lauder. A Solution to the Flowgraphs Case Study using Triple Graph Grammars and eMoflon. In Pieter Van Gorp, Louis M. Rose, and Christian Krause, editors, *Sixth Transformation Tool Contest, TTC 2013*, volume 135 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 69–74, 2013. (Cited on Page 69.)
- [3] Anthony Anjorin, Marius Lauder, Michael Schlereth, and Andy Schürr. Support for Bidirectional Model-to-Text Transformations. In Jordi Cabot, Tony Clark, Manuel Clavel, and Martin Gogolla, editors, *10th Workshop on OCL and Textual Modelling, OCL 2010*, volume 36 of *Electronic Communications of the EASST (ECEASST)*. European Association for the Study of Science and Technology (EASST), 2010. (Cited on Page 55.)
- [4] Anthony Anjorin, Marius Lauder, Sven Patzina, and Andy Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. In *Informatik 2011*, volume 192 of *Lecture Notes in Informatik (LNI)*, page 281. Gesellschaft für Informatik (GI), 2011. (Cited on Pages 209 and 210.)
- [5] Anthony Anjorin, Karsten Saller, Sebastian Rose, and Andy Schürr. A Framework for Bidirectional Model-to-Platform Transformations. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012*, volume 7745 of *Lecture Notes in Computer Science (LNCS)*, pages 124–143. Springer, 2012. (Cited on Pages 55 and 224.)
- [6] Anthony Anjorin, Andy Schürr, and Gabriele Taentzer. Construction of Integrity Preserving Triple Graph Grammars. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations - 6th International Conference, ICGT 2012*, volume 7562 of *Lecture Notes in Computer Science (LNCS)*, pages 356–370. Springer, 2012. (Cited on Pages 114, 138, 147, 185, 187, and 239.)
- [7] Anthony Anjorin, Gergely Varró, and Andy Schürr. Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques. In Frank Hermann and Janis Voigtländer, editors, *First International Workshop on Bidirectional Transformations, BX 2012*, volume 49 of *Electronic Communications of the EASST (ECEASST)*. European Association for the Study of Science and Technology (EASST), 2012. (Cited on Pages 69 and 234.)

- [8] Anthony Anjorin, Erhan Leblebici, Andy Schürr, and Gabriele Taentzer. A Static Analysis of Non-Confluent Triple Graph Grammars for Efficient Model Transformation. In Holger Giese and Barbara König, editors, *Graph Transformation - 7th International Conference, ICGT 2014, Held as Part of STAF 2014*, volume 8571 of *Lecture Notes in Computer Science (LNCS)*, pages 130–145. Springer, 2014. (Cited on Pages 185, 198, 218, 239, and 241.)
- [9] Anthony Anjorin, Sebastian Rose, Frederik Deckwerth, and Andy Schürr. Efficient Model Synchronization with View Triple Graph Grammars. In Jordi Cabot and Julia Rubin, editors, *Modelling Foundations and Applications - 10th European Conference, ECMFA 2014, Held as Part of STAF 2014*, volume 8569 of *Lecture Notes in Computer Science (LNCS)*, pages 1–17. Springer, 2014. (Cited on Page 230.)
- [10] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. Modularizing Triple Graph Grammars Using Rule Refinement. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*, volume 8411 of *Lecture Notes in Computer Science (LNCS)*, pages 340–354. Springer, 2014. (Cited on Pages 69, 97, and 234.)
- [11] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MoDELS 2010*, volume 6394 of *Lecture Notes in Computer Science (LNCS)*, pages 121–135. Springer, 2010. (Cited on Pages 213 and 241.)
- [12] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, September 2003. ISSN 0740-7459. (Cited on Page 33.)
- [13] Holger Bock Axelsen and Robert Glück. What Do Reversible Programs Compute? In Martin Hofmann, editor, *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011*, volume 6604 of *Lecture Notes in Computer Science (LNCS)*, pages 42–56. Springer, 2011. (Cited on Page 229.)
- [14] Jean Bézivin. On the Unification Power of Models. *Software and Systems Modeling (SoSyM)*, 4(2):171–188, 2005. (Cited on Pages 1, 9, and 211.)
- [15] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *16th IEEE International Conference on Automated Software Engineering, ASE 2001*, volume 872565, pages 273–280. IEEE Computer Society, 2001. (Cited on Page 33.)
- [16] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In Krzysztof

- Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008*, volume 5301 of *Lecture Notes in Computer Science (LNCS)*, pages 53–67. Springer, 2008. (Cited on Page 85.)
- [17] Enrico Biermann, Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Gabriele Taentzer. Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation. In Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel, editors, *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, volume 5765 of *Lecture Notes in Computer Science (LNCS)*, pages 121–140. Springer, 2010. (Cited on Pages 237 and 245.)
- [18] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang : Resourceful Lenses for String Data. In George C. Necula and Philip Wadler, editors, *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 407–419. ACM, 2008. (Cited on Page 230.)
- [19] Manuel Bork, Leif Geiger, Christian Schneider, and Albert Zündorf. Towards Roundtrip Engineering - A Template-Based Reverse Engineering Approach. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008*, volume 5095 of *Lecture Notes in Computer Science (LNCS)*, pages 33–47. Springer, 2008. (Cited on Page 226.)
- [20] Howard Chivers and Richard Paige. XRound: Bidirectional Transformations and Unifications Via a Reversible Template Language. In *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005*, volume 3748 of *Lecture Notes in Computer Science (LNCS)*, pages 205–219. Springer, 2005. (Cited on Page 226.)
- [21] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006. (Cited on Pages 35 and 224.)
- [22] Krzysztof Czarnecki, John Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In Richard F. Paige, editor, *Theory and Practice of Model Transformations, Second International Conference, ICMT 2009*, volume 5563 of *Lecture Notes in Computer Science (LNCS)*, pages 260–283. Springer, 2009. (Cited on Pages 51, 224, 229, and 233.)
- [23] Duc-Hanh Dang and Martin Gogolla. On Integrating OCL and Triple Graph Grammars. In Michel Chaudron, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2008*, volume 5421 of *Lecture Notes in Computer Science (LNCS)*, pages 124–137. Springer, 2009. (Cited on Pages 234 and 240.)

- [24] Frederik Deckwerth and Gergely Varró. Attribute Handling for Generating Preconditions from Graph Constraints. In Holger Giese and Barbara König, editors, *Graph Transformation - 7th International Conference, ICGT 2014, Held as Part of STAF 2014*, volume 8571 of *Lecture Notes in Computer Science (LNCS)*, pages 81–96. Springer, July 2014. (Cited on Pages 198, 200, 205, and 213.)
- [25] DIN 8580:2003-09. Manufacturing Processes - Terms and Definitions, Division, 2003. (Cited on Page 2.)
- [26] Zinovy Diskin. Algebraic Models for Bidirectional Model Synchronization. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruehl, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008*, volume 5301 of *Lecture Notes in Computer Science (LNCS)*, pages 21–36. Springer, 2008. (Cited on Page 230.)
- [27] Sven Efftinge. oAW xText: A Framework for Textual DSLs. In *Workshop on Modeling Symposium at Eclipse*, 2006. (Cited on Page 225.)
- [28] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. ISBN 978-3-540-31187-4. (Cited on Pages 13, 17, 24, 25, 26, 27, 28, 35, 39, 70, 74, 77, 84, 111, 113, 128, 131, 132, 134, 147, 185, 194, 235, and 236.)
- [29] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. Information Preserving Bidirectional Model Transformations. In Matthew Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007*, volume 4422 of *Lecture Notes in Computer Science (LNCS)*, pages 72–86. Springer, 2007. ISBN 978-3-540-71288-6. (Cited on Pages 121, 129, 132, 134, and 240.)
- [30] Hartmut Ehrig, Claudia Ermel, and Frank Hermann. On the Relationship of Model Transformations Based on Triple and Plain Graph Grammars. In *Third International Workshop on Graph and Model Transformations, GRaMoT 2008*, pages 9–16. ACM, 2008. (Cited on Page 86.)
- [31] Hartmut Ehrig, Claudia Ermel, Frank Hermann, and Ulrike Prange. On-the-Fly Construction, Correctness and Completeness of Model Transformations Based on Triple Graph Grammars. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems, 12th International Conference, MoDELS 2009*, volume 5795 of *Lecture Notes in Computer Science (LNCS)*, pages 241–255. Springer, 2009. (Cited on Page 240.)
- [32] Hartmut Ehrig, Frank Hermann, and Christoph Sartorius. Completeness and Correctness of Model Transformations Based on Triple Graph Grammars with Negative Application Conditions. In Artur Boronat and Reiko Heckel, editors, *8th International Workshop on Graph Transformation and Visual Modeling Techniques, GT-VMT 2009*, volume 18 of *Electronic Communications*

- of the EASST (ECEASST). European Association for the Study of Science and Technology (EASST), 2009. (Cited on Pages 121, 128, 131, and 240.)
- [33] Steven D. Eppinger. Model-Based Approaches to Managing Concurrent Engineering. *Journal of Engineering Design*, 2(4):283–290, 1991. (Cited on Page 2.)
 - [34] Romina Eramo and Alessio Bucaioni. Understanding Bidirectional Transformations with TGGs and JTL. *Second International Workshop on Bidirectional Transformations, BX 2013*, 57, 2013. (Cited on Page 232.)
 - [35] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations, 6th International Workshop, TAGT 1998*, volume 1764 of *Lecture Notes in Computer Science (LNCS)*, pages 157–167. Springer, 2000. (Cited on Page 212.)
 - [36] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. ISBN 0-201-48567-2. (Cited on Page 210.)
 - [37] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 1 edition, 2010. ISBN 0321712943. (Cited on Pages 12 and 63.)
 - [38] Leif Geiger, Thomas Buchmann, and Alexander Dotor. EMF Code Generation with Fujaba. In *5th International Fujaba Days*, 2007. (Cited on Page 212.)
 - [39] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and Analysis using GROOVE. *STTT*, 14(1):15–40, 2012. (Cited on Page 241.)
 - [40] Holger Giese and Stephan Hildebrandt. Efficient Model Synchronization of Large-Scale Models. Technical report, Hasso-Plattner Institute at the University of Potsdam, 2009. (Cited on Pages 235 and 237.)
 - [41] Holger Giese, Stephan Hildebrandt, and Stefan Neumann. Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel, editors, *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, volume 5765 of *Lecture Notes in Computer Science (LNCS)*, pages 555–579. Springer, 2010. (Cited on Pages 12 and 246.)
 - [42] Holger Giese, Stephan Hildebrandt, and Leen Lambers. Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars - Ensuring Conformance of Relational Model Transformation Specifications and Implementations. *Software and Systems Modeling (SoSyM)*, 13(1):273–299, 2014. (Cited on Pages 240 and 241.)

- [43] Ulrike Golas, Hartmut Ehrig, and Frank Hermann. Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions. In Rachid Echahed, Annegret Habel, and Mohamed Mosbah, editors, *Third International Workshop on Graph Computation Models, GCM 2010*, volume 39 of *Electronic Communications of the EASST (ECEASST)*. European Association of Software Science and Technology (EASST), 2011. (Cited on Pages 235 and 240.)
- [44] Thomas Goldschmidt and Steffen Becker. Classification of Concrete Textual Syntax Mapping Approaches. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008*, volume 5095 of *Lecture Notes in Computer Science (LNCS)*, pages 169–184. Springer, 2010. (Cited on Page 225.)
- [45] Joel Greenyer and Ekkart Kindler. Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars. *Software and Systems Modeling (SoSyM)*, 9(1):21–46, 2010. (Cited on Pages 12 and 231.)
- [46] Joel Greenyer and Jan Rieke. Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *Applications of Graph Transformations with Industrial Relevance - 4th International Symposium, AGTIVE 2011*, volume 7233 of *Lecture Notes in Computer Science (LNCS)*, pages 222–237. Springer, 2012. (Cited on Pages 12, 236, 237, 239, and 241.)
- [47] Joel Greenyer, Sebastian Pook, and Jan Rieke. Preventing Information Loss in Incremental Model Synchronization by Reusing Elements. In Robert France, Jochen Kuester, Behzad Bordbar, and Richard Paige, editors, *Modelling Foundations and Applications - 7th European Conference, ECMFA 2011*, volume 6698 of *Lecture Notes in Computer Science (LNCS)*, pages 144–159. Berlin / Heidelberg, 2011. Springer. (Cited on Page 238.)
- [48] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering, ICSE 2008*, pages 925–926. ACM, 2008. (Cited on Page 225.)
- [49] Esther Guerra, Juan De Lara, and Fernando Orejas. Pattern-Based Model-to-Model Transformation: Handling Attribute Conditions. In Richard F. Paige, editor, *Theory and Practice of Model Transformations, Second International Conference, ICMT 2009*, volume 5563 of *Lecture Notes in Computer Science (LNCS)*, pages 83–99. Springer, 2009. (Cited on Page 234.)
- [50] Esther Guerra, Juan de Lara, Dimitrios Kolovos, and Richard Paige. Inter-Modelling: From Theory to Practice. In Dorina Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MoDELS 2010*, volume 6394 of *Lecture*

- Notes in Computer Science (LNCS)*, pages 376–391. Springer, 2010. (Cited on Page 232.)
- [51] Annegret Habel and Karl-heinz Pennemann. Correctness of High-Level Transformation Systems Relative to Nested Conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009. (Cited on Pages 205 and 207.)
 - [52] Frank Hermann, Hartmut Ehrig, Fernando Orejas, and Ulrike Golas. Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Graph Transformations - 5th International Conference, ICGT 2010*, volume 6372 of *Lecture Notes in Computer Science (LNCS)*, pages 155–170. Springer, 2010. (Cited on Page 240.)
 - [53] Frank Hermann, Ulrike Golas, and Fernando Orejas. Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In Jean Bézivin, Mark Richard Soley, and Antonio Vallecillo, editors, *First International Workshop on Model Driven Interoperability MDI 2010, held in conjunction with MoDELS 2010*, pages 22–31, Oslo, Norway, 2010. ACM. ISBN 9781450302920. (Cited on Pages 240 and 241.)
 - [54] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of Model Synchronization Based on Triple Graph Grammars. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems, 14th International Conference, MoDELS 2011*, volume 6981 of *Lecture Notes in Computer Science (LNCS)*, pages 668–682. Springer, 2011. (Cited on Page 238.)
 - [55] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. In Perry Alexander, Corina Pasareanu, and John Hosking, editors, *26th International Conference on Automated Software Engineering, ASE 2011*, pages 480–483. IEEE Computer Society, 2011. ISBN 9781457716393. (Cited on Page 229.)
 - [56] Stephan Hildebrandt, Leen Lambers, Holger Giese, Dominic Petrick, and Ingo Richter. Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *Applications of Graph Transformations with Industrial Relevance - 4th International Symposium, AGTIVE 2011*, volume 7233 of *Lecture Notes in Computer Science (LNCS)*, pages 238–253. Springer, 2011. (Cited on Pages 202 and 231.)
 - [57] Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. A Survey of Triple Graph Grammar Tools. In Perdita Stevens and James Terwilliger, editors, *Second International Workshop on Bidirectional Transformations, BX 2013*, volume 57 of *Electronic Communications of the EASST (ECE-ASST)*. European Association for the Study of Science and Technology (EASST), 2013. (Cited on Page 233.)

- [58] Berthold Hoffmann, Dirk Janssens, and Niels Van Eetvelde. Cloning and Expanding Graph Transformation Rules for Refactoring. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 152:53 – 67, 2006. ISSN 15710661. (Cited on Page 237.)
- [59] Ákos Horváth, Gergely Varró, and Dániel Varró. Generic Search Plans for Matching Advanced Graph Patterns. In Karsten Ehrig and Holger Giese, editors, *Sixth International Workshop on Graph Transformation and Visual Modeling Techniques, GT-VMT 2007*, volume 6 of *Electronic Communications of the EASST (ECEASST)*. European Association for the Study of Science and Technology (EASST), 2007. (Cited on Pages 91 and 235.)
- [60] Lennart C. L. Kats and Eelco Visser. The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463. ACM, 2010. (Cited on Page 225.)
- [61] Rick Kazman, Steven S. Woods, and S. Jeromy Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *5th Working Conference on Reverse Engineering, WCRE 1998*, pages 154–163. IEEE Computer Society, 1998. (Cited on Page 228.)
- [62] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In Perry Alexander, Corina S Pasareanu, and John G Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011*, pages 163–172. IEEE Computer Society, 2011. (Cited on Page 215.)
- [63] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. Consistency-Preserving Edit Scripts in Model Versioning. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*, pages 191–201. IEEE Computer Society, 2013. (Cited on Page 215.)
- [64] Ekkart Kindler and Robert Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007. (Cited on Page 234.)
- [65] Ekkart Kindler, Vladimir Rubin, and Robert Wagner. An Adaptable TGG Interpreter for In-Memory Model Transformations. In Andy Schürr and Albert Zündorf, editors, *Second International Fujaba Days*, pages 35–38, 2004. (Cited on Page 235.)
- [66] Felix Klar, Alexander Königs, and Andy Schürr. Model Transformation in the Large. In Ivica Crnkovic and Antonia Bertolino, editors, *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT*

- Symposium on the Foundations of Software Engineering, ESEC-FSE 2007*, pages 285–294. ACM, 2007. ISBN 9781595938114. (Cited on Pages 236 and 237.)
- [67] Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel, editors, *Graph Transformations and Model Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, volume 5765 of *Lecture Notes in Computer Science (LNCS)*, pages 141–174. Springer, 2010. (Cited on Pages 63, 121, 192, 239, and 241.)
- [68] Alexander Königs. *Model Integration and Transformation - A Triple Graph Grammar-Based QVT Implementation*. PhD thesis, Technische Universität Darmstadt, 2008. (Cited on Page 234.)
- [69] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Conflict Detection for Graph Transformation with Negative Application Conditions. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations, Third International Conference, ICGT 2006*, volume 4178 of *Lecture Notes in Computer Science (LNCS)*, pages 61–76. Springer, 2006. (Cited on Pages 194, 195, and 205.)
- [70] Marius Lauder. *Incremental Model Synchronization with Precedence-Driven Triple Graph Grammars*. PhD thesis, Technische Universität Darmstadt, 2013. (Cited on Pages 183, 212, 239, and 246.)
- [71] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Efficient Model Synchronization with Precedence Triple Graph Grammars. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations - 6th International Conference, ICGT 2012*, volume 7562 of *Lecture Notes in Computer Science (LNCS)*, pages 401–415. Springer, 2012. (Cited on Pages 45, 121, 137, and 241.)
- [72] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Developing eMoflon with eMoflon. In Davide Di Ruscio and Dániel Varró, editors, *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014*, volume 8568 of *Lecture Notes in Computer Science (LNCS)*, pages 138–145. Springer Verlag, Springer, 2014. (Cited on Pages 210 and 214.)
- [73] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. A Catalogue of Optimization Techniques for Triple Graph Grammars. In Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer, editors, *Modellierung 2014*, volume 225 of *Lecture Notes in Informatics (LNI)*, pages 225–240. Gesellschaft für Informatik, Gesellschaft für Informatik (GI), 2014. (Cited on Pages 67 and 235.)
- [74] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A Comparison of Incremental Triple Graph Grammar Tools. In Frank Hermann and Stefan Sauer, editors, *13th International Workshop on Graph Transformation and Visual Modeling Techniques, GT-VMT 2014*, volume 67 of *Electronic Communications of EASST (ECEASST)*.

- European Assoc. of Software Science and Technology, European Association for the Study of Science and Technology (EASST), 2014. (Cited on Pages 233, 237, and 238.)
- [75] Nuno Macedo, Tiago Guimarães, and Alcino Cunha. Model Repair and Transformation with Echo. In Ewen Denney, Tefvik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*, pages 694–697. IEEE, 2013. (Cited on Page 232.)
- [76] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 152:125–142, 2006. ISSN 15710661. (Cited on Pages 35 and 224.)
- [77] Manfred Nagl. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. Springer, 1996. (Cited on Page 227.)
- [78] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. Xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002. ISSN 15335399. (Cited on Page 232.)
- [79] Fernando Orejas and Elvira Pino. Correctness of Incremental Model Synchronization with Triple Graph Grammars. In Davide Di Ruscio and Dániel Varró, editors, *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014*, volume 8568 of *Lecture Notes in Computer Science (LNCS)*, pages 74–90. Springer, 2014. (Cited on Page 239.)
- [80] Terence John Parr. Enforcing Strict Model-View Separation in Template Engines. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *13th Conference on World Wide Web, WWW 2004*. ACM, 2004. (Cited on Page 58.)
- [81] Terence John Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007. ISBN 0978739256. (Cited on Pages 56, 58, 62, 63, 214, 217, and 224.)
- [82] Terence John Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf, 2009. ISBN 978-1-93435-645-6. (Cited on Pages 56, 58, 62, and 214.)
- [83] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1 edition, 1991. ISBN 0262660717. (Cited on Pages 17 and 19.)
- [84] Terrence W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *J. Comput. Syst. Sci.*, 5(6):560–595, 1971. (Cited on Page 231.)
- [85] Sebastian Rose, Marius Lauder, Michael Schlereth, and Andy Schürr. A Multidimensional Approach for Concurrent Model Driven Automation Engineering. In Janis Osis and Erika Asnina, editors, *Model-Driven Domain*

- Analysis and Software Development: Architectures and Functions*, pages 90–113. IGI Publishing, 2011. (Cited on Pages 12 and 240.)
- [86] David E. Rydeheard and Rod M. Burstall. *Computational Category Theory*. Prentice Hall, 1988. ISBN 978-0-13-162736-9. (Cited on Pages 13, 17, 18, and 72.)
- [87] Andy Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: formale Definitionen, Anwendungsbeispiele und Werkzeugunterstützung*. DUV: Informatik. Deutscher Universitäts-Verlag (Springer Fachmedien Wiesbaden), 1991. ISBN 978-3-8244-2021-6. (Cited on Pages 91 and 235.)
- [88] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG 1994*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163. Springer, 1994. (Cited on Pages 12, 121, 129, 132, 231, and 240.)
- [89] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations, 4th International Conference, ICGT 2008*, volume 5214 of *Lecture Notes in Computer Science (LNCS)*, pages 411–425. Springer, 2008. (Cited on Pages 239 and 240.)
- [90] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2. edition, 2009. ISBN 978-0-321-33188-5. (Cited on Pages 34 and 85.)
- [91] Perdita Stevens. A Landscape of Bidirectional Model Transformations. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007*, volume 5235 of *Lecture Notes in Computer Science (LNCS)*, pages 408–424. Springer, 2008. (Cited on Pages 51 and 229.)
- [92] Perdita Stevens. Towards an Algebraic Theory of Bidirectional Transformations. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations, 4th International Conference, ICGT 2008*, volume 5214 of *Lecture Notes in Computer Science (LNCS)*, pages 1–17. Springer, 2008. (Cited on Pages 229 and 240.)
- [93] Perdita Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software and Systems Modeling (SoSyM)*, 9(1): 7–20, 2010. (Cited on Pages 231 and 234.)
- [94] Perdita Stevens. Bidirectionally Tolerating Inconsistency: Partial Transformations. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*, volume 8411 of *Lecture Notes in Computer Science (LNCS)*, pages 32–46. Springer, 2014. (Cited on Pages 45 and 244.)

- [95] Gabriele Taentzer and Arend Rensink. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005*, volume 3442 of *Lecture Notes in Computer Science (LNCS)*, pages 64–79. Springer, 2005. (Cited on Page 85.)
- [96] Gergely Varró and Frederik Deckwerth. A Rete Network Construction Algorithm for Incremental Pattern Matching. In Keith Duddy and Gerti Kappel, editors, *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013*, volume 7909 of *Lecture Notes in Computer Science (LNCS)*, pages 125–140. Springer, 2013. ISBN 9783642388835. (Cited on Pages 212, 218, and 235.)
- [97] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C L Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. ISBN 978-1-4812-1858-0. (Cited on Pages 33 and 63.)
- [98] Robert Wagner. *Inkrementelle Modellsynchronisation*. PhD thesis, Universität Paderborn, 2009. (Cited on Page 234.)
- [99] Martin Wieber, Anthony Anjorin, and Andy Schürr. On the Usage of TGGs for Automated Model Transformation Testing. In Davide Di Ruscio and Dániel Varró, editors, *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014*, volume 8568 of *Lecture Notes in Computer Science (LNCS)*, pages 1–16. Springer, 2014. (Cited on Pages 202 and 231.)
- [100] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitris Kolovos, Richard Paige, Marius Lauder, and Andy Schürr. A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations - 4th International Conference, ICMT 2011*, volume 6707 of *Lecture Notes in Computer Science (LNCS)*, pages 31–46. Springer, 2011. (Cited on Page 236.)
- [101] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a Reversible Functional Language. In Alexis De Vos and Robert Wille, editors, *Reversible Computation - Third International Workshop, RC 2011*, volume 7165 of *Lecture Notes in Computer Science (LNCS)*, pages 14–29. Springer, 2011. (Cited on Page 229.)

ACRONYMS

UML Unified Modeling Language

OMG Object Management Group

OO Object-Oriented

MDE Model-Driven Engineering

MOF Meta-Object Facility

MDA Model-Driven Architecture

CME Concurrent Manufacturing Engineering

QVT Query View Transformation

TGG Triple Graph Grammar

BX Bidirectional Transformation

GPL General Purpose Language

DSL Domain Specific Language

OCL Object Constraint Language

CAD Computer Aided Design

CAM Computer Aided Manufacturing

CNC Computerized Numerical Control

CLS Cutter Location Source

MPF Main Program File

TIE Tool Integration Environment

EMF Eclipse Modeling Framework

XML eXtensible Markup Language

MOCA eMoflon Code Adapter

API Application Programming Interface

JSON JavaScript Object Notation

EBNF Extended Backus-Naur Form

EMOFLON A metamodeling and model transformation tool (www.emoflon.org)

NAC Negative Application Condition

CPA Critical Pair Analysis

JMI Java Metadata Interfaces

EMOF Essential MOF

EA Enterprise Architect

CASE Computer Aided Software Engineering

DEC Dangling Edge Check

JET Java Emitter Templates

CURRICULUM VITAE



Personal Details

Date of Birth	01. 11. 1984
Place of Birth	Zaria, Nigeria
Nationality	German, Nigerian

Work Experience

5/2010–5/2015	PhD Thesis, research associate and teaching assistant at the Real Time Systems Lab, Technische Universität Darmstadt
10/2007–3/2008	Software Developer, MAN-Nutzfahrzeuge AG, Munich
6/2007–9/2007	Software Developer, REA-Elektronik GmbH, Darmstadt
12/2002–9/2003	Civil Service (Zivildienst), Evangelisches Hospital für palliative Medizin, Frankfurt

Education

4/2008–4/2010	Master of Science, Computational Engineering, Technische Universität Darmstadt
10/2003–4/2007	Bachelor of Science, Computational Engineering, Technische Universität Darmstadt
9/2001–6/2002	Feststellungsprüfung, Studienkolleg für ausländische Studierende, Mittel-Hessen
11/1994–6/2000	West African Senior School Certificate, King's College Lagos, Nigeria